To: Jaguar Hardware ERS Distribution
From: Toby Farrand
Date: 13 November 1989
Re: Review of Jaguar Hardware ERS

One of the keys to Jaguar's success is its ability to specify and freeze certain design decisions as early as possible in the design process. Because of the long lead times of hardware design in general, and the high level of technical complexity of the Jaguar design in particular, it is critical that we gain broad agreement on a stable foundation of technical design choices, so as to prevent later changes in direction which cause delays in the development of the product. A further strategy for managing the design complexity of Jaguar is to carefully modularize and document the overall design so that smaller, more manageable groups can work with a high level of autonomy on the design.

The key tool for implementing this strategy is the hardware External Reference Specification (ERS) of the machine. The goal of this document is to define as accurately as possible, all of the inputs and outputs and relationships of the different aspects of the machine, so that each module can be designed without a huge amount of interaction between all of the groups. The danger of any modular design is that opportunities for global optimization of the design can be missed. To alleviate this problem, the Jaguar team is inviting a small number of people from across Apple Products to review the preliminary hardware ERS, and have a chance to influence the design decisions. Our goal is to have the design reviewed and finalized by mid-January. This requires that all input on the hardware design be received by early December.

The Jaguar team would like to invite you to review our preliminary hardware ERS. This is an opportunity that isn't without its obligations.

As I am sure you will appreciate, it is very important that this document remain secure. Your hardware ERS is numbered and registered in your name. Please do not copy the ERS.

Because we are limiting the number of ERS copies that are going outside the group, we would like to make sure you, or a person from your group reads and gives us feedback on the document. If you cannot or do not wish to participate in the review process, then please let us know so that we can pass your copy on to someone who can give us the feedback we need to make the Jaguar machine the best possible machine. The time to change anything in the Jaguar hardware design is now. It is imperative that any feedback you have on the proposed design be given as soon as possible. In the next month, the Jaguar team will be holding a series of design reviews to try to encourage your feedback on the hardware ERS. Because of the complexity of the Jaguar design, the design reviews will not be able to go into great detail on the design. These will mostly be used as forums to encourage discussion of any controversial points in the design. Written comments and feedback directly to individuals on the Jaguar team are quite welcome.

In short, by giving you a hardware ERS, we are asking you to participate in a meaningful way in the success of the Jaguar machine. We expect that this will require a non-trivial amount of your time. We appreciate and wish to be responsive to your feedback.

Apple Forward

External Reference Specification
Special Projects

November 15, 1989

Hugh Martin, Director Special Projects
Phone: x4-4302
AppleLink: MARTIN
MS: 60-AA

Dear Reader,

Thank you for taking time to study this first section of the Jaguar System ERS. It represents a great deal of thought and effort by the team to define an architecture and system that can be the basis for a long-lived family of machines.

I would like to set the context for this document by reviewing Jaguar's goals and principle directions. Our overriding goal is to:

"Develop a new family of machines that will extend the markets already pioneered by the Macintosh."

We will do this by developing a revolutionary baseline for personal computing that will spark the imaginations of developers and customers. Jaguar will be the most exciting personal computer you can buy for the desktop!

A tall order..... how will we do it?

Jaguar is maniacally focused on the user. Though the computer is very powerful, it will seem extremely personal. The technologies embodied in the architecture are there primarily because they can be used to contribute to the user's experience.

From the users perspective, Jaguar will certainly do all things accomplished by personal computers today, only faster. In addition, his/her interaction with the system will be different in three new ways:

Integrated Media

Jaguar is as fluent with digital video and audio as Macintosh is with bit-mapped graphics. All video and audio capabilities are fully integrated into both the system hardware and software. The system is not only capable of editing and playing back integrated media, it can synthesize it as well. In hardware, this means Jaguar can draw 2D and 3D images at animation rates. In software, the graphics primitives are provided, as well as tools to generate scenes and control movement. These capabilities mean that Jaguars will be superb simulation and visualization engines.

Digital Assistant

Jaguar will fundamentally change the way you work with computers to a manner that more naturally extends your endeavors. Your interactions with Jaguar will be much as if you had an "assistant". Documents will be stored and retrieved based on content and context ("Show me the documents I've received from Ed recently"). Digital Assistant can be customized to the user's wishes ("Make this folder contain links concerning move to Bandley 3"). The user interface becomes a dynamic, active, and more intelligent tool.

The Global Desktop

Jaguar is designed to be used at a distance through the integral ISDN or phone connection. Limited speech recognition and natural language processing will allow remote voice control of the system ("FAX me the '89 cost spreadsheet at 212-424-2300"). Jaguar will also have a media-rich integrated mail system which will allow mail to be sent and received in the most convenient format.

The Jaguar Family

There are currently three Jaguar family members envisioned: low end, mid-range, high-end. The family is planned to range from $2,500 to $15,000 ASP. The mid-range model is the first product out the door and is the product commonly referred to as Jaguar. Its cost is $1,700 to $1,900 w/o monitor. The low and high-end models are being planned to at least the block level to assure we do achieve a scalable family.

Every Jaguar in the family will be capable of supporting the previously mentioned functionality. This means all Jaguars have at a minimum: lots of Mips and MFlops, digital signal processing, integral phone and ISDN connections, sound in/out, and video decompression hardware. The attached Hardware ERS describes in detail the first implementation of these and other capabilities.

The Jaguar Software ERS will be published approximately May 1. Other sections to be also completed include: monitors, packaging, and power supply.

Once again, thank you for your interest and time.

Apple Jaguar Hardware ERS

External Reference Specification
Special Projects

November 15, 1989

Ron Hochsprung, ERS Coordinator
Phone: x4-2661
AppleLink: HOCHSPRUNG1
MS: 60-AA

This External Reference Specification presents the design of the electronics hardware for the first Jaguar product (referred to as Jag1).

The **Jaguar Hardware ERS** is the first of several ERSs which will be forthcoming to describe the base product. Among these other ERSs are:

Jaguar Software ERS
Jaguar Monitor ERS
Jaguar Power Supply ERS
Jaguar Product Design ERS

In addition to the base machine, several expansion cards are anticipated to be available at product introduction. These will be described in additional ERSs. Several cards have been discussed:

Jaguar Video Capture (and Compression?) card
Jaguar Network cards

Table of Contents

Jaguar Hardware
External Reference Specification
Special Projects

15 November 1989

This Table of Contents represents the entire Jaguar Hardware ERS document. Note that each chapter's page numbers include an acronym of the chapters contents. This will allow us to update the chapters without changing the global numbering of others.

Jaguar Hardware Overview (JHO)

This chapter introduces the key components of the Jag1 implementation.

Jaguar CPU (CPU)

This chapter introduces some of the important concepts which are contained within the processor (Motorola MC88110) used for Jaguar.

Memory (MEM)

This chapter describes the Main Memory, Frame Buffer and System Controller for Jag1.

Jaguar Graphics Overview (JGO)

This chapter describes the overall architecture of Graphics support in Jaguar. It provides the introduction for the Video Back End, Video Decompression and Expansion and Wilson chapters.

Video Back End (VBE)

This chapter deals with the output side of the Frame Buffer and how data gets processed for both RGB and Video (NTSC/PAL) monitors.

Video Decompression (VDC)

This chapter deals with the hardware which will support real-time decompression of video data.

Expansion and Wilson (E&W)

This chapter deals with the Window DMA architecture and implementation for Jag1.

Bus Like Thing (BLT)

This chapter deals with the expansion bus of Jaguar.

I/O Subsystem (I/O)

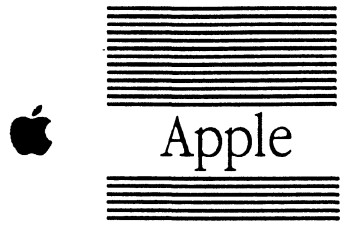
This chapter introduces the concepts and implementation of the I/O subsystem for Jaguar.

System Issues (SYS)

This chapter deals with several topics which cut across multiple subsystems within the Jag1.

Manufacturing Issues (MFG)

This chapter deals with new manufacturing issues introduced by the Jaguar implementation strategy.



Apple Jaguar Hardware Overview

External Reference Specification
Special Projects

Nov. 15, 1989

Return Comments to:
Ron Hochsprung
Phone: x4-2661
AppleLink: HOCHSPRUNG1
MS: 60-AA

144

Apple CONFIDENTIAL

Contents

Jaguar Hardware Introduction

The purpose of this chapter is to set the stage for the reader as to what Jaguar Hardware is all about and what makes it different from existing personal computers. Following chapters will present each of the functional blocks (introduced here) in great detail.

Ultimately, a separate "Jaguar Family Architecture" document will be produced. In the mean time, this Hardware ERS will attempt to serve both as an introduction to the family in addition to its stated purpose of describing the first member of that family. In order to make the distinction between areas in which we are discussing the family versus the member, we will use the term ***Jaguar*** to mean the family, while ***Jag1*** will refer to the particular first system.

The Jaguar family presents a revolutionary set of functionalities to the personal computer world in a manner analogous to the way in which the original Macintosh did. While enhancing the Mac's traditional features (e.g., bit-mapped graphics, WYSIWYG, direct manipulation user interface), the Jaguar adds capabilities designed to support new classes of applications which embody combinations of three areas:

Real Time Animation/Simulation

While the Macintosh allows some degree of "what if" applications (e.g., spreadsheets), the base level of performance forces applications to deal with "static" objects only. For example, when a spreadsheet cell's value is changed, the re-calculation of the rest of the spreadsheet is palpable. If a graph is being displayed from the spreadsheet, a very noticeable "compute" time is observed before the graph changes.

The level of performance offered by the Jaguar system allows simulation of "dynamic" objects. As a simple example, imagine that a "slider" is associated with a spreadsheet's cell. As the slider is moved, the spreadsheet is seen to be directly tracking the slider's movements. The graph is updated in "real time", in direct response to the user's input.

New graphics and load /store instructions in the processor and the high-bandwidth system interconnect and memory designs of the Jaguar yield the performance necessary to do "real time" 3D animation. The result of all of this attention to performance makes the Jaguar a machine which is as adept in a 3D world as the Mac has been in 2D.

Connectivity

The most obvious implement with which a person interacts on a daily basis is the telephone. Yet, no direct access is available to today's personal computer user. Jaguar changes this by making the telephone an integral part of the user's computer world. By providing direct support for telephone connectivity (for both analog and ISDN phone systems), Jaguar enables new classes of applications.

Besides giving the basic connectivity (i.e. getting the signals into the box), the Jaguar's performance allows the CPU to perform tasks which have traditionally required specialized circuitry (e.g., a

modem)or special processors (e.g., DSPs). The base system is capable of performing complete modem functions by using the general purpose CPU. For example, Jaguar can do all the processing necessary to become a FAX modem, without specialized hardware. This flexibility allows adding new services (e.g., supporting a new modem protocol) by a simple software module.

Integrated Media

Another class of data which has yet to be effectively tapped by existing personal computers is the growing world of video. Jaguar introduces video as an inherent data type which can be captured, compressed, manipulated, stored and displayed. For example, instead of pasting a static graphics object into a document, a video object would be used. The video would play back as the document was later displayed.

In addition to video, Jaguar also enhances the utility of audio data. In addition to just playback, Jaguar also includes hardware to capture audio data; even the voice input coming in over the telephone provides a source of audio. The hardware also includes Sample Rate Conversion hardware to ease the task of integrating multiple sources of sound, not all of which were sampled at the same rate.

Note that the performance level of Jaguar's 3D graphics allows computer generated animation to be yet another medium which can be integrated with the live video and sound.

Raising the Base

While adding features to directly support the above, it should be apparent that the Jaguar is also a superior system for existing applications; its base level of performance will be unmatched. Thus, the Jaguar family also enables applications which require the much higher level performance base of the Jaguar family.

New Concepts in Jaguar

The Jaguar has been designed with several new features which may be unfamiliar to the Macintosh world. These concepts are discussed briefly here to set the stage for more complete discussions in the various chapters of this ERS.

Intelligent I/O

The Mac has traditionally been designed with the assumption that the CPU will be used to implement all I/O transfers. Hardware support for most I/O has been extremely minimal, usually just enough to access a device (e.g., SCC). The drivers for each device (and/or protocol) must deal with all sorts of vagaries to make up for the minimalist hardware.

Newer models of the Mac have added some hardware support for such things as LocalTalk (e.g., the PIC) and SCSI block transfers (e.g., SCSI-DMA chip). However, this hardware has been done on a piece-meal basis, with little affect on the overall I/O model.

The Jaguar I/O model introduces a consistent approach to I/O; it does this by presenting a single abstraction of Channel Programs for all devices. Channel Programs are data structures which specify a sequence of actions, many of which are simple data transfers,to be performed. They are interpreted by an "I/O Engine" (Wankel) which manages all of the low-level details of device interactions. The corresponding data transfers are managed by DMA channels within the I/O chip (Mazda).

The goal of this abstraction is to allow subsequent implementations of Jaguars to change the underlying hardware (based on price/performance) without changing the view of the I/O system as seem from the software (e.g., the Pink Access Managers).

By off-loading the CPU(s) from managing I/O at a very low level, the Jaguar greatly increases the efficiency of multi-threaded systems, such as Pink.

Window DMA (Graphics Streams)

Jaguar includes hardware designed to support block transfers of graphics data between maim memory and the frame buffer. This directly supports the concept of "back buffering" which is part of the Pink (more specifically, Albert) model for graphics.

At the simplest level, it would seem that all that is required is to be able to move blocks of data from one place to another; i.e., a simple DMA. However, transferring "windows" from one place to another is not a simple DMA operation. DMA normally deals with a contiguous stream of data from one place (or device) to another. Windows are not contiguous areas! Little details like "row bytes", etc. make normal DMA's fairly useless (or, at least, very cumbersome) for window transfers. In

addition, there are some simple transformations which could be done on the data (e.g., clip masking) which do not fit the simple DMA model, but which offer large improvements in performance if they could be done in hardware.

For this reason, Jaguar's "Window DMA" adds semantics to the fetching and storing of data which is aware of the "windowness" of the sources and destinations of transfers. Such transfers are viewed as consisting of three logically distinct pieces: the sourcing of data (which may involve "windowing"), the transfer of the data (which is always viewed as a stream of data) and the sinking of the data (again, possibly by windowing).

In addition to simple transfers, Jaguar's Window DMA system allows several transformations of these graphics data streams. At the abstract level, a "data flow" model is presented. Data can be processed by "piping" source streams of data into transformation blocks. The output of these blocks is just another stream, which can proceed to other blocks until, ultimately, the resulting stream is placed into a destination (window).

A simple example of how this mechanism comes in handy in Jaguar would be the display of "frame grabbed" video, sourced by an expansion card, being displayed into a window in the frame buffer. The frame grab card merely needs to produce a stream of data. That is, it does not need to know anything about the ultimate destination (e.g., the window location, row bytes, etc.); it simply produces a stream of data (sequence of pixels, with no associated addressing). The window DMA services will place this stream into the frame buffer, under control of a mask stream.

Video Decompression

In many potential uses of Jaguar, "video" is an important data type. While the display of live video in a window is useful, it is only a small piece of the media integration. Video data must be able to be stored and edited so that it can be incorporated into documents as easily as graphics is today. However, raw video data is huge!! Without some form of compression, the resources of mass storage and networking would be severely overtaxed.

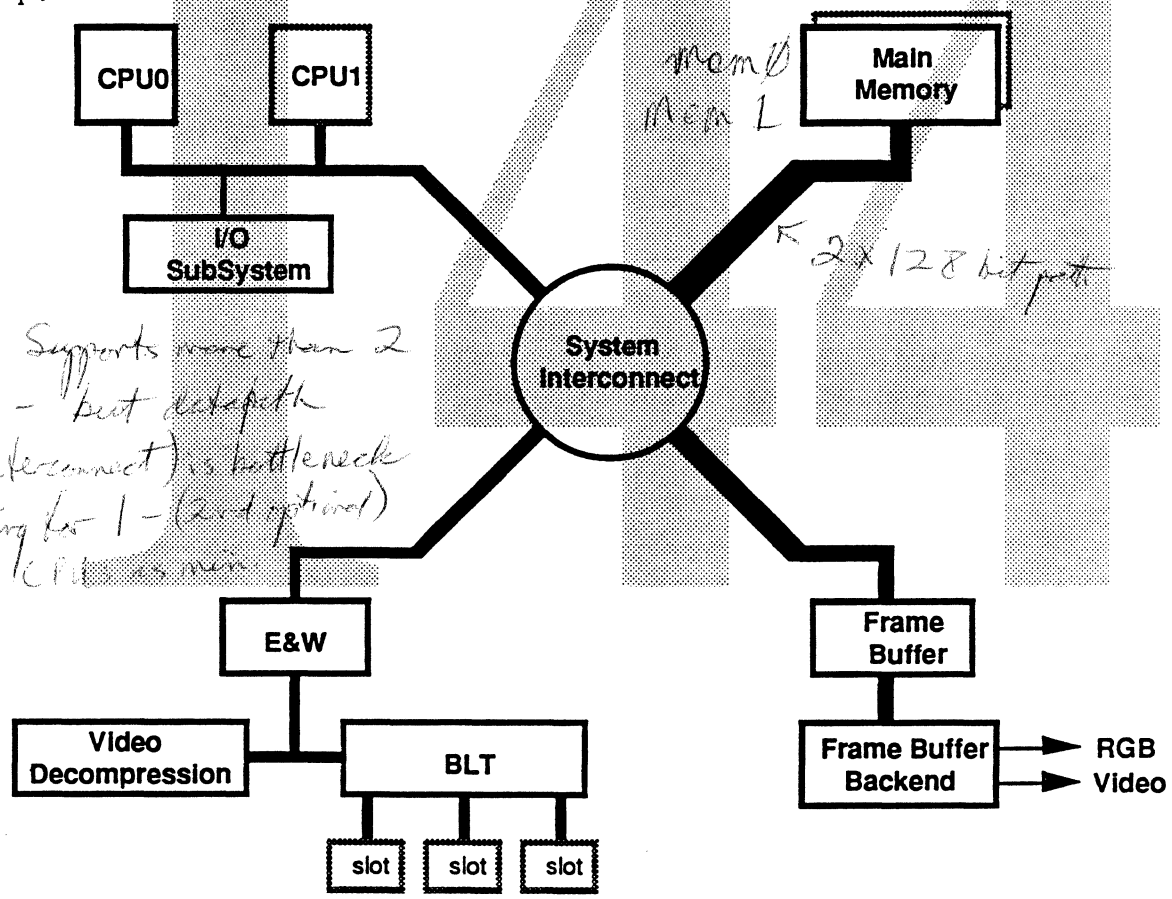
Fortunately, several video compression standards are emerging which offer a tremendous reduction in the resources required to store video data. These standards allow, for example, realistic video tele-conferencing with transmission bandwidth requirements which are within reason for current LANs and/or ISDN; or, for video sequences to be played back from a hard disc (or, MO drive) in real-time.

Jaguar will include hardware support for these standards. This is very important for any system which claims to provide integrated media. The Jag1 will have hardware designed for real-time decompression ; the same hardware can also be used for non-real-time compression. Real-time compression (especially from a "live" video source) will be supported by add in cards on BLT. For details, see the section on Video Decompression.

Jag1 Hardware Overview

In the following sections, we will introduce the major hardware components of the Jag1 system. Since each section is described in great detail in the remainder of this ERS, the descriptions contained here are minimal. The intent is to give the reader a basic understanding of the parts and how they fit together.

The System Block Diagram , which is a highly abstracted view of the hardware, shows the main functional blocks and the associated data paths between them. (Note that each block does not necessarily correspond to a single chip.)



System Block Diagram

The following overview of the hardware is organized to correspond to the above diagram.

CPU(s)

The main processor for the Jaguar is a new version of the Motorola 88000 family which has been enhanced (with input from Jaguar's team) in several areas over the existing implementation. This processor (which will be the MC88110) will be referred to as XJS in the ERS. The CPU chapter contains more information about the XJS features.

I/O Subsystem

Since there are many components involved in the I/O Subsystem, the System Block Diagram does not go into any detail of this block.; this section merely introduces the major I/O devices which form the subsystem. See the I/O Subsystem chapter for a more detailed diagram of the I/O devices and how they interconnect.

The I/O Subsystem of Jag1 is composed of many elements. While they are described as if they are totally independent, some of these elements may share hardware and/or software. The following devices and/or connectivity is provided in the base Jag1.

Low Cost Hard Disk (LCHD)

Jag1 will have an on-board 80 MB hard disc. This disk is primarily designed to be used as the Opus Residence and swapping disc for Pink's virtual memory system.

High Capacity Floppy

The floppy disc for Jag1 will support all floppy formats; this includes all of Apple's plus industry standard (i.e., IBM). In the Apple format, the disc capacity will be up to 5 MB.

SCSI

In addition to the standard LCHD, additional discs (and other devices) can be added via the traditional SCSI route. The SCSI support for Jag1, however, will include synchronous transfers to give Jag1 a higher throughput for SCSI transfers. This, along with the intelligence of Wankel, will result in a disc system which has remarkably higher performance than current systems.

EtherNet

Every Jaguar will include an EtherNet port. The external connection scheme (Apple's "FriendlyNet" standard) allows any of the standard EtherNet interconnects to be used (Thick, Thin and Twisted Pair). Because of the intelligent I/O engine and DMA, this EtherNet interface is high performance without requiring significant CPU resources.

LocalTalk/Asynchronous Serial

While EtherNet provides high-bandwidth LAN support, Jag1 will provide a LocalTalk port, which can double as a normal serial port. Since the LLAP layer is supported directly by Wankel, LocalTalk does not interfere with CPU activity.

ISDN

Jaguar is designed to standardly support the evolving world of ISDN. As with the other communications connections, most of the low-level protocol handling will be performed by Wankel.

Analog Phone

For those user's who can not take advantage of ISDN (or, for those who want an additional "phone" connection), a standard analog connection is provided. Note that the processing power of the CPUs will be used to provide modem support (e.g., V.22 and FAX) via the analog connection.

Sound Support

Sound is a very important medium in the Jaguar environment. To support sound effectively, the Jag1 will include "CD quality" Stereo input (ADC's) and output (DAC's). Additional support is provided for telephone rates via a separate CODEC.

ADB

Apple Desktop Bus peripherals (e.g., keyboard and mouse) are supported via an ADB port. The ADB protocol is managed directly by Mazda, again requiring little CPU intervention.

Desktop Interconnect

When a Jaguar implementation is physically partitioned into two "boxes", connectivity for the I/O with which the user interacts most directly (e.g., Sound in/out, ADB) must be placed on the "desktop". In such implementations, Jaguar will provide a high-speed communications path between the "main" box and the "desktop". This interconnect will be fast enough to support all of the sound and ADB traffic.

Main Memory

The Main Memory of Jag1 is composed of two banks of 128-bit wide DRAM . The base configuration, with just 1 bank populated, will be 8 MB (with a possibility of a 4 MB configuration, depending upon DRAM configurations), which can be extended with an additional bank of DRAM SIMMs. Using 16 Mb parts, the Jag1 can grow to a 128 MB machine.

Optional parity checking on Main Memory is supported. Parity is done on a byte basis, thus requiring an additional 16 bits of DRAM. When parity checking is enabled, data path chips within

the System Interconnect will generate parity on writes and check parity on reads.

System Interconnect

The System Interconnect provides a common path for data between the several masters (CPU(s), Mazda, E&W) and the two logical banks of memory. It provides sufficient buffering and control to allow several memory transactions to be in progress at one time. For example, while memory is busy performing a read from CPU0, a write transaction can be acted upon from CPU1.

Frame Buffer

The Frame Buffer is composed of a 96-bit wide, 1.5 MB VRAM bank. This provides sufficient memory (and bandwidth) to support the standard Jaguar monitors. The random access ports of the VRAMs are connected to the System Interconnect; the serial ports connect to the Video Back End. No parity option is provided for the Frame Buffer.

Video Back End

The serial ports of the Frame Buffer's VRAMs connect to the Video Back End sub-section. This block provides gamma correction, video convolution, etc. with output ports for both RGB and video (NTSC, PAL).

E&W

E&W is the chip which encapsulates two different, although related, functions. One part of E&W (the W, for the Wilson) implements support for graphics and/or video streams ("Window DMA"). The second function (the E, for Expansion) provides the basic data and control paths for accesses between the processors (CPUs and Mazda) and BLT slots; note that this also includes access between BLT slots and the memories.

BLT (Bus Like Thing)

The BLT sub-system provides a high-bandwidth (~ 300 MB/s) pathway for expansion cards. This high bandwidth is necessary for real time manipulation of data streams at video rates. Organized as a packet-switched cross-bar, BLT allows multiple transactions to proceed in parallel. While providing this performance, the interface presented to the card designer is simple. Thus, a simple card can be implemented with minimal hardware.

Note that while all possible interactions are supported (e.g., Reads and Writes from the CPUs), BLT and E&W are optimized to provide high-bandwidth transfers from BLT slots to memory. For example, the maximum bandwidth is provided for data transfers from a video Frame Grab card to the Frame Buffer.

Video Decompression

This block performs the computations and data movement to support several different compression standards. It connects onto the BLT side of E&W and functions as a "virtual slot" with respect to data transfer requests to the System Interconnect.

Expansion Options

The hardware described in this ERS represents the "base" system. However, the goal of the Jaguar team is to have expansion capabilities available at the time of introduction. This section discusses some of these expected options.

High Capacity Hard Disc

The built-in hard disc is designed for swapping and some amount of system storage. While 80 MB may be large by today's standards, it will clearly be inadequate for the volumes of data (even with compression) which will be required for video editing, etc. One of the standard options will be a "big" hard disc (contained within the system unit).

Of course, since SCSI will be available as a system connector, external drives may also be added to any system.

Magneto Optic (MO) Drive

An MO drive would be an alternative (or addition) to a hard disc. This drive would provide a large capacity, removable media storage element to the Jaguar.

Digital Audio Tape (DAT) Drive

Several DAT drives are already available for use as data (i.e., non-audio) storage. These devices allow Gigabytes of storage on a small, cheap medium.

Video Input Card

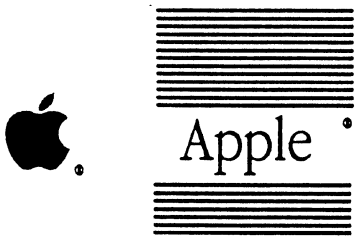
The base Jaguar does not provide direct video input. The model is that video will come from "servers", the network or mass storage (e.g., MO-drive). For users who need a source of live video, a BLT expansion card will provide this service.

Real Time Compression

The Video Decompression hardware in the base Jaguar can support non-real-time compression of video and/or audio data. To allow complete symmetric compression (e.g., for authoring), a BLT expansion card will be provided. Since the video source may often consist of "live" video, a single card which combines Video Input with Compression would make sense.

Network Support

While the base Jaguar includes ISDN, LocalTalk and EtherNet, some users may require additional network (e.g., TokenRing) access. The simplicity of the BLT card interface should make the "porting" of network cards to the Jaguar a fairly simple and straightforward process.



Jaguar CPU

External Reference Specification
Special Projects

Nov. 15, 1989

Return Comments to:
Ron Hochsprung
Phone: x4-2661
AppleLink: HOCHSPRUNG1
MS: 60-AA

144

Contents

Introduction

This chapter discusses the main features of the Jaguar processors (XJS). This chapter is not meant to be a complete processor description; it deals primarily with those features which with the reader should be familiar to understand some of the other Jag1 design chapters.

The Motorola 88110 (XJS)

The Motorola 88110 is the second-generation of their 88K RISC (Reduced Instruction Set Computer). It remains compatible to the first generation 88100/88200 (at the user program level), but adds many new features (or, extends prior ones) which make it an ideal processor for Jaguar. The following features are incorporated within the XJS:

Multiple Independent Function Units

The CPU is implemented as a set of independent function units. Each unit is capable of acting upon its inputs and producing its output(s) simultaneously. The following units are present: 2 integer units, a Load/Store unit, a Floating Point ADD unit, a Floating Point MUL unit, a Floating Point DIV unit and 2 Graphics units.

Note that the Floating Point units are designed to support full 80-bit IEEE format operands. They Floating Point Multiply and Divide units also handle the integer multiply and divide instructions, respectively.

Dual-Instruction Issue

The primary advantage of the multiple units is that several instructions can be processed in parallel. Depending upon the availability of data (and the state of the functional units required), 2 instructions can be issued during every clock. While data dependencies will reduce the throughput somewhat, the average execution rate is expected to be close to 2 instructions per clock for optimized code; this means that the peak execution rate approaches 100 Mips (assuming a 50 MHz clock).

Unlike some other processors, the programmer does not have follow any special coding rules to "force" the processor to perform this optimization. The hardware will always attempt to issue two instructions per clock! Of course, utilizing this feature to the utmost will require that the compilers be careful in their instruction generation. However, since no special encodings are required, logically correct code will always run.

General Register File (GRF)

As in the original 88K model, there is a general purpose register set of 32 32-bit registers which are used for integer and graphics operations, addressing, bit-field operations, etc. (While the XJS has a separate register set for Floating Point operands, the older style of 88K floating point instructions (which utilize the general register file) are still supported for compatibility.)

Extended Register File (XRF)

A second register set (32 x 80 bits) is provided for floating point computations. Besides adding the necessary width to support full Double Extended floating point formats, the additional registers free up the integer registers for use by pointers, indices, etc. which allow more overlap of floating point and integer operations in highly optimized code.

This extra register set is especially suited for some of our graphics transformation processing, where the transformation matrix values can reside entirely in the register set while still providing scratchpad space for the actual computations.

Control Register File (CRF)

The GRF and XRF are accessible by user code. Another (mostly) only accessible by system code is used to present and/or maintain processor state. Of these, 5 are allocated for use by the operating system; they are not used by the hardware. The most important use for such a register is to maintain any "processor unique" data (e.g., its "id" and its system stack pointer).

Fast Floating Point

The Floating Point units of the XJS have been re-designed and optimized over those of the originaal 88K. The Floating Point ADD and MUL units are capable of producing their results in 3 clocks. Since each unit is able to initiate a new operation every clock, the peak floating point rate (at 50 MHz) is 100 MFLOPs (i.e., both a FADD and FMUL being issued each clock)!

The XRF provides the large register set necessary to fully utilize these pipelined, parallel functional units; it also provides the storage for the full IEEE extended data type.

> Note: While use of the GRF for Floating Point has been maintained for compatiblity, Jaguar's Floating Point computations will use the XRF!

Graphics Instructions

A set of instructions is provided which allows operations on several pixels simultaneously, thus greatly improving the graphics bandwidth for algorithms which can take advantage of them. The graphics instructions can manipulate multiple pixels contained in a 64-bit unit in parallel. The graphics unit

enhancement makes many rendering algorithms much faster than possible using normal instructions on individual pixels.

Note that since the graphics instructions are supported by independent units, they are subject to the same 2 instruction issue speedup as normal integer operations.

On-chip Caches

Two 8 KB caches (32-byte lines x 128 sets x 2-way associativity) are contained on-chip, one each for Instructions and Data. The CPU is thus not as memory bandwidth limited (depending upon the characteristics of the program) as would normally be the case.

The caches are "physical" (s opposed to "virtual"). That is, the cache tags are the physical addresses of their contents. This results if fewer (if any) explicit cache flushing required when performing a task context switch as compared to virtual caches.

The data cache can be used in two modes (selectable on a page basis): CopyBack or WriteThrough . In WriteThrough, writes which hit in the cache will update both the cache and memory; i.e., the memory is always kept consistent because all writes cause memory updates.

In CopyBack, a write will only update the cache's copy. If the cache line corresponding to the write's target is not currently in the cache, that line will be brought into the cache first, then that cache line is modified with the written data. However, no external write is performed. Instead, the modified cache line will be marked "dirty". Any dirty cache line which is reused will be written back to memory before the new data is read.

Jaguar assumes the CopyBack mode will be used almost exclusively. This is because the CopyBack can reduce memory traffic substantially. For example, during a normal procedure call, registers are saved on a software managed stack. If one assumes that all the leaf routines will tend to use the same memory locations for this stack area, then it would be possible for the no memory writes (or, their read backs) to be required for the duration of a programs execution. All of the stack traffic would be entirely on chip!

Cache Coherency Protocol

However, in order to effectively use CopyBack caches in a multi-processor environment, the CPUs must implement a coherency protocol which guarantees that all processors see consistent, correct copies of data, some of which may be present in the various caches. How the XJS implements this is discussed in more detail in the Cache Coherency section of this chapter.

Branch Target Cache

In a pipelined processor (like XJS), branches occurring in code can appreciably affect performance because the processor is stalled while waiting for the new instructions to be fetched. In order to alleviate this bottleneck, the XJS includes a Branch Target Cache (BTC).

The purpose of the BTC is to allow fetching of instructions at the target of the branch while the normal instruction sequencer is fetching the sequential instructions. The XJS's BTC contains 32 fully-associative entries which are able to store 2 instructions each. The BTC is accessed by the address of a branch instruction. By the time that the decision is known, both the sequential and target instructions are ready; the condition simply chooses the appropriate set.

After the correct path has been determined, the normal instruction sequencing will obtain the correct "follow on" instructions beyond those chosen by the BTC (or, normal sequence).

Address Translation Hardware

The XJS supports a demand paged virtual memory system (like Opus) by providing hardware translation from virtual to physical addresses. The model provides a two-level address space with 4 KB page size.

A 32 entry translation cache is provided for each of the instruction and data paths. These translation caches are fully associative and provide very high hit rates, thus minimizing the memory bandwidth required for fetching page table entries.

If a virtual address is not found in the translation caches, the XJS will execute an entirely hardware managed "table walk". The resulting page table entry will then replace the least recently used translation cache entry.

In order to support shared pages efficiently, the table walk mechanism provides for "indirect" page table entries. That is, the normally final entry in a page table can actually contain the address of the real page table entry. The operating system can then maintain only one page table for shared data structures, instead of manipulating each sharer's page table when the state of a shared page changes (e.g., it gets swapped out).

Block Address Translation

For some parts of the address space (e.g., frame buffer or I/O), it becomes cumbersome to partition the area into pages. The XJS provides 8 Block Address Translation Cache entries for each of the instruction and data units. Entries in the BATCs can map areas up to 4 MB.

Interrupts

The XJS has only two interrupt input pins. One of these (INT*) is used for normal interrupts processing. The hardware includes disable control to prevent nested interrupts until the system has saved enough state to allow them. This INT* line is the one which is controlled by Mazda as part of the interrupt system.

Another input (NMI*) is primarily designed for debugging. This Non-Maskable Interrupt is always allowed to interrupt the processor, even when the state of the processor is not recoverable! The primary intention of this interrupt is to allow analysis of "system hangs", where the system may be non-interruptable (i.e., via INT*).

XJS Bus Interface

Because of its speed, the XJS requires a high-bandwidth memory interface with which to keep it "fed". Several features have been supplied in the XJS bus protocol which can allow an implementation (e.g., Jag1) to optimize bus/memory bandwidth.

64-bit Data Path

The XJS's data interface is 64 bits wide. The chip (and its bus protocol) allow data to be transferred at one 64-bit wide double word each clock (e.g., 50 MHz). This wide path allows an entire cache line to be transferred in as little as 5 clocks (including the address presentation).

Address Pipelining

The XJS Bus separates the action of specifying the address and function code (the request) from the transfer of the corresponding data (the response). In effect, the bus is logically divided into two busses: the Address Bus and the Data Bus.

The bus protocol allows the Address bus to be used somewhat independently from the Data bus. In other words, one processor (or other XJS Bus Master) can present its request (on the Address bus) while a prior transaction's response data is being transferred over the Data bus. This capability is called "Address Pipelining".

The XJS uses Address Pipelining between its own Instruction and Data units.

By adding somewhat more complex logic external to the XJS, a complete "split transaction" protocol can be implemented. That is, the request for a transaction (which uses the Address bus) can logically be separated from that transaction's response (which requires the Data bus).

The protocol allows multiple Data busses to be used in a system, as long as a single Address bus is used (for cache coherency reasons, as described below). Jag1 plans to use this split response mechanism, with multiple Data busses to effect the maximum utilization of memory bandwidth between the various bus masters in the system (XJSs, Mazda and E&W).

Cache Coherency

As mentioned in the XJS description above, the XJS implements a cache coherency protocol for maintaining coherency between XJS caches and memory. The operating system has control over those areas of the address space which it desires to be kept coherent. It does so by means of a

"Global" bit in the page table entries. Only Global data is maintained using the cache coherency mechanism which works as follows:

All XJSs must be connected to a single Address Bus. Whenever a Global request is made (via the Address bus), all XJSs will "snoop" their data caches to ascertain if they have a modified copy of the requested cache line. If one does (and, it will be the only one if everyone is playing by the rules), it signals the requestor, which will then back off the bus and try again later.

In the mean time, the processor which snoop "hit" will write its modified cache line back to memory and mark the line as clean within its cache.

When the original requestor retries the access, it will get the correct, updated memory image.

Whenever an XJS does a read of a cache line as part of a write, it uses a special function code which means "read with intent to modify". This code is used by all processors to invalidate any copy of that line which it may have in its cache (assuming that its copy is not "dirty", which performs as above) .

When an XJS attempts to dirty a clean line in its cache, it will make a bus transaction using a special function code which means "invalidate". This will cause all other XJSs to invalidate any local copy which they may have.

This protocol guarantees that their is at most one modified version of a cache line among any number of XJSs. Any access by other XJSs to that line will force a memory update and a subsequent read.

The System Controller follows a set of rules which guarantees that this coherency mechanism works, even with the split transactions and multiple data busses.

> Software Note: Careful attention to task (thread) migration issues should be made. Indiscriminant use of coherency (e.g., threads accessing common data running in multiple CPUs) can have serious performance impacts on system resources.

Apple Jaguar Memory

External Reference Specification
Special Projects
Jaguar

November 14, 1989

Return Comments to:
Tony Masterson
Spencer Worley
Phone: x4-6414,8469
AppleLink: MASTERSON1,SPENCER.W
MS: 60D

Contents

Section 1

Introduction

Goals of real-time animation and integrated media such as live video requires Jaguar to have a high-performance memory system. A live 640X480 non-interlaced NTSC video window at 60 frames/sec chews up to 98 MB/sec of frame buffer bandwidth. A half screen of back buffered animation at 15 frames/sec takes 16 MB/sec of DRAM bandwidth, and 16 MB/sec of frame buffer bandwidth. Rendering polygons can easily take 125 to 200 MB/sec of DRAM bandwidth for the Z-buffer values, and the pixels. Running typical applications on the XJS takes around 48 MB/sec of DRAM bandwidth. If you try to do all four on a two processor Jag1, you have peak requirements of 264 MB/s of DRAM bandwidth, and 114 MB/s of frame buffer bandwidth. In order to support these requirements , we have made changes to the traditional memory design of a PC. For example, the Jag1 memory system allows simultaneous access to the frame buffer and DRAM. Reads and writes are queued up into buffers. Point-to-point interconnect is used to reduce the capacitance that each device must drive, giving a smaller off chip delay. All of these features are implemented by using a silicon Datapath to connect everything together. It should be noted that Jag1 does not meet the peak requirements just described, as the description is really meant to justify the cost and complexity of our memory subsystem.

The Jag1 memory subsystem consists of 1 or 2 processors, one or two 128 bit wide banks of DRAM, one 96 bit wide bank of VRAM, the Datapath connecting everything together, two clock synchronizers, and a system controller. The system is designed to run at 40 MHz and 50 MHz. The following is a description of each of the major blocks of the subsystem followed by a description of the hardware.

System Memory

The system memory consists of either one or two banks of 128 bit wide DRAM with an optional 16 bits of parity. The memory appears to software as 64 bit wide memory, but is implemented as 128 bit wide banks in order to increase the bandwidth. Each SIMM used for memory is 36 bits wide, consisting of 32 bits of data plus 4 bits of optional byte parity, and uses 72 pins. Fast page mode 80 ns access DRAM parts are used on the SIMMs. The four SIMM types supported are:

- 1 Mbyte data using either two 256KX16 or eight 256KX4 DRAMs and 128 Kbyte of parity using four 256KX1 DRAMs
- 2 Mbyte data and 256 Kbyte parity using four 512KX8 DRAMs and four 512KX1 DRAMs
- 4 Mbyte data and 512 Kbyte parity using eight 1MX4 DRAMs and four 1MX1 DRAMs

- 16 Mbyte data and 2 Mbyte parity using eight 4MX4 DRAMs and four 4MX1 DRAMs

The memory configurations possible using the four SIMM types are:

Number of 1 MByte SIMMs in system	Number of 2 MByte SIMMs in system	Number of 4 MByte SIMMs in system	Number of 16 MByte SIMMs in system	Total amount of system memory in megabytes
4				4
8	or 4			8
4	4			12
	8	or 4		16
4		4		20
	4	4		24
		8		32
			4	64
4			4	68
	4		4	72
		4	4	80
			8	128

The memory bandwidth will depend on the processor clock speed. The table below summarizes the bandwidth for cache line reads and writes:

Memory type and operation	Bandwidth for 40 MHz	Bandwidth for 50 MHz
DRAM Read	128 MB/sec	145 MB/sec
DRAM Write	160 MB/sec	160 MB/sec

Parity

Parity is generated on a byte basis for all writes to the DRAM, and will only detect errors, not correct them. To eliminate any performance penalty for using parity, the parity is generated for the write

data as it passes through the Datapath ASICs that are between the DRAM and the rest of the system. For reads, parity is generated for the data as it passes through the Datapath, but is also compared with the parity value read from the parity DRAM. If the two do not match, and parity checking is enabled, then a transaction error is returned to the master requestor. Parity errors can be forced to allow the parity circuit to be tested. Parity can be disabled to reduce the cost of the system by eliminating the need for parity DRAM on the SIMMs.

Frame Buffer

Jag1 has built-in either a 1.5 megabyte frame buffer consisting of 12 128KX8 VRAM chips or a 3 megabyte frame buffer consisting of 24 256KX4 VRAM chips. Fast page mode 100 ns VRAM parts are used. The random access port for the VRAM is connected to the Datapath, and the shift port is connected to the video backend ASIC (refer to the graphics subsystem for a full description of the video back-end). The 1.5 megabyte frame buffer will support any monitor containing up to .5 million 24 bit pixels, or up to 1 million 8 bit pixels. The 3 megabyte frame buffer will support up to 1 million 24 bit pixels, or up to 2 million 8 bit pixels.

There is a one bit per pixel mask plane placed in the frame buffer for PAL and NTSC modes that selects what areas of the screen get convolved, so that live video windows can be displayed at full resolution without convolution. In addition, a 64X64 non-convolved, or a 16X16 convolved 24BPP cursor is placed in the frame buffer and is supported in hardware.

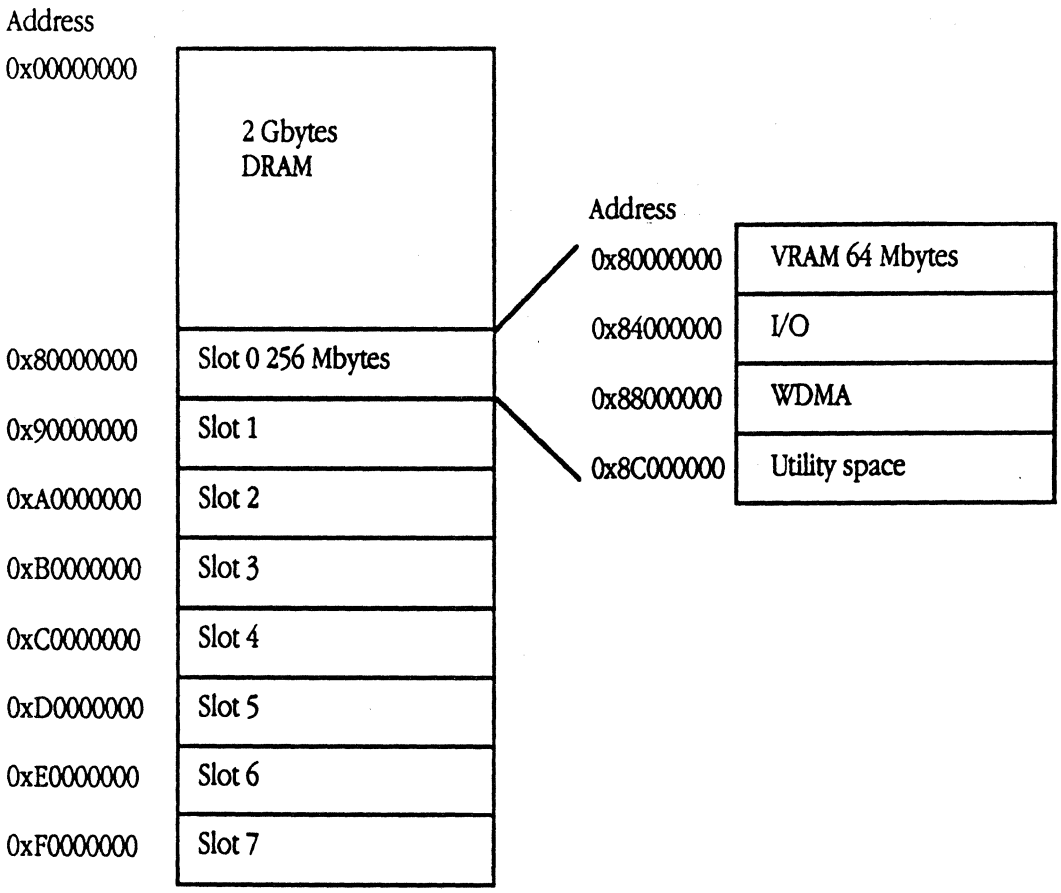
The random access port of the frame buffer is configured differently for 8 bit/pixel(BPP), 24 BPP, and convolved 24 BPP display modes. For 24 BPP mode, the frame buffer is implemented as a single 1.5 megabyte 96 bit wide bank of VRAM, but appears to software as a 48 bit wide memory. For convolved 24 BPP the frame buffer works as two 48 bit wide banks of VRAM with one bank holding the even line pairs, and the other holding the odd line pairs, but still appears to software as a single 48 bit wide bank. For an explanation of convolution see the video backend ERS. For 8 BPP mode, only 1 megabyte of VRAM is visible in order to make the frame buffer appear as a single 64 bit wide memory to software. Two Datapath ASICs connecting the VRAM to the rest of the system implement the different display modes.

The frame buffer bandwidth for cache line reads and writes depends on the processor speed, and is summarized below:

Memory type and operation	Bandwidth for 40 MHz	Bandwidth for 50 MHz
VRAM Read	98 MB/sec	107 MB/sec
VRAM Write	107 MB/sec	114 MB/sec

Memory map

The memory map for the Jaguar architecture is:



Slot 0 to 7 refer to expansion card slots on BLT, however slot 0 is reserved for system space and for local card space. Utility space is used to provide each card with a fixed address to read its slot ID from. Both DRAM and VRAM are aliased throughout their address spaces. Refer to the I/O subsystem and the graphics subsystem chapters for further decoding of the I/O and WDMA address blocks.

Datapath

Having a silicon Datapath in Jag1 allows us to easily perform the following functions:

1) Allow frame buffer accesses to happen simultaneously with DRAM accesses.

2) Generate and check parity on-the-fly for the DRAM.

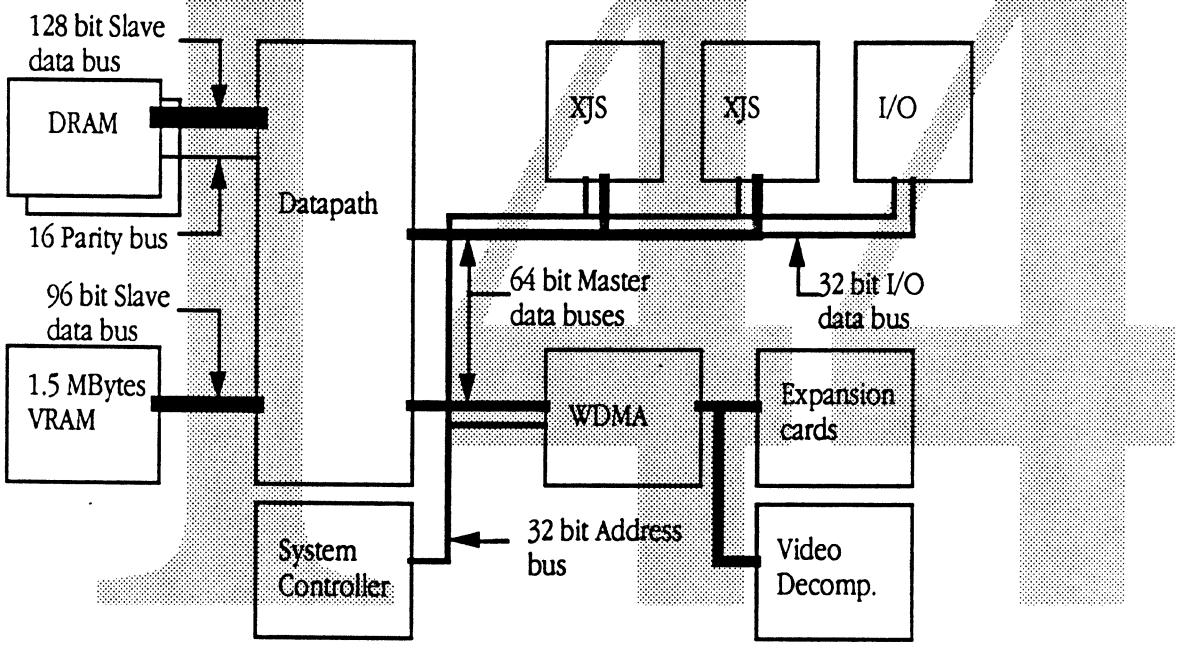
3) Reconfigure the VRAM for the different screen modes and depths.

4) Reduce system capacitance

5) Allow the I/O ASIC to use only a 32 bit databus

6) Perform read and write buffering

Ideally each master and slave device would have its own data bus into the Datapath to allow simultaneous use of all resources. However, due to pin limitations and complexity issues, the Datapath masters are divided between two master busses going into the Datapath. Since the expansion port and the window DMA engine both require a significant part of the bandwidth, they are placed on their own bus, while the 88110s and the I/O ASIC share the other bus. This allows the processors to be running out of DRAM, while the window DMA engine writes a video window to the VRAM. Both buses use the 88110 split transaction bus protocol for master transactions, and the I/O and WDMA ASIC additionally support slave transactions. The read and write buffering in the Datapath allows the masters to transmit and receive data out of arbitration order. However, to ensure cache coherency with minimal hardware, operations must be in order to each slave device. Additionally, the 88110 protocol requires all transaction to be performed in order to each master. Below is a block diagram of the memory subsystem.



Since the I/O ASIC uses only a 32 bit data bus, the Datapath ASICs in the system convert all I/O master operations from 32 bit to 64 bit.

System Controller

The System Controller ASIC oversees all master and slave transactions feeding into and out of the Datapath, as well as transfers between the I/O system and the CPUs. Cache coherency for the CPU caches is maintained at all times using the 88110 snoop protocol.

The System Controller queues up to 4 requests. If a write is requested, and the write buffer for the requested slave device is available, then the data bus is granted to that master, the write data is immediately placed into the buffer, and the master gets an acknowledge. When the slave device requested is free, the data is written to the device from the buffer. For reads, when the slave device requested is free, the data bus is granted to the requestor, and the data from the slave is passes to the master.

There are two contraints that the System Controller enforces. To easily support cache coherency, all transactions to a slave device must be performed in order of address arbitration. To follow the 88110 protocol, all transactions for each master must be done in order. Internally the System Controller will keep a queue for each constraint, and coordinate them. The System Controller can perform operations out of order as long as the two contraints are met.

When the system is powered up, parity will be disabled, and there will be 4K bytes of DRAM available.

Section 2

Hardware Implementation

The following is a detailed description of the hardware components of the memory and processor subsystem. There are two different ASICs used to implement it, the System Controller and the Datapath Chip. The Datapath ASIC is slice of the datapath and two are used per system.

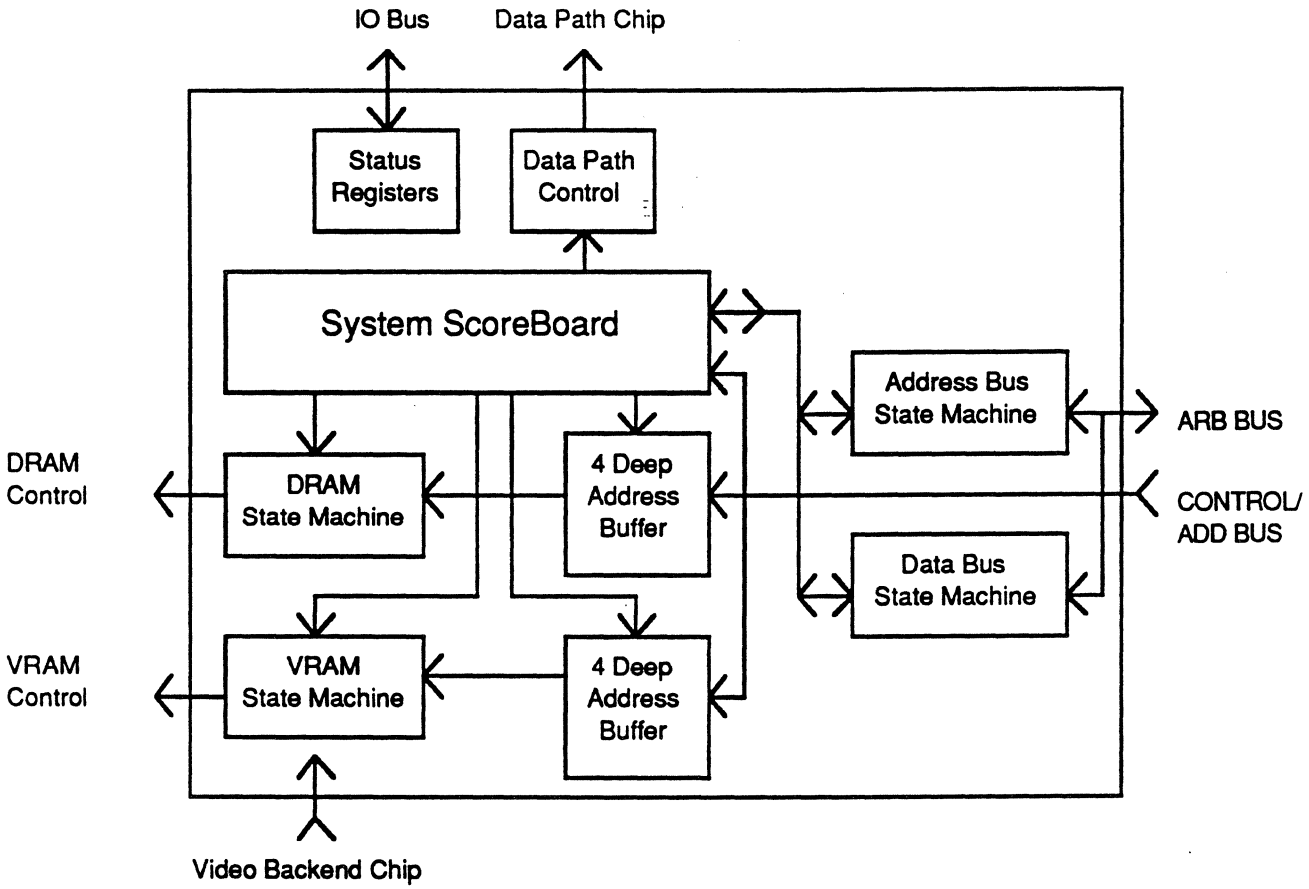
System Controller

The System Controller (SC) has 7 interfaces:

- XJS Address Bus
- Video Backend Chip (Elmer)
- DRAM Control
- IO System Low Speed Bus
- XJS Arbitration Bus
- DataPath Chip Control
- VRAM Control

The SC controls the data accesses for the XJS chips, the E & W chip (Expansion interface and Wilson), and the IO Chip (Mazda). The SC also provides an interface to allow the Video Backend Chip (Elmer) to perform transfer cycles. In addition, the SC controls the DataPath Chip (DPC), the DRAMs and the VRAMs. The IO System Low Speed Bus, allows software access to the status and control registers on the SC.

The SC supports the interface specified in the XJS Bus Specification, operating as a slave on this bus. Requests come from the 2 CPUs, Mazda, and Wilson. Mazda and Wilson can also act as slaves. In this case the SC coordinates the data transfers between these slaves and their masters. In all cases the SC will acknowledge the address and control the data transfer using special protocols defined in this document. All resources can have a total of 4 requests queued for them in the System Scoreboard before stalling the address bus. For the VRAM and DRAM, four corresponding addresses are saved. All writes are acknowledged as soon as the data is stored in the write buffer contained in the DataPath Chip. Resource management and data ordering is maintained by a scoreboard on the SC. Data ordering is maintained such that requests to any slave will complete in order of being issued across all masters. Also, requests by each master will complete in the order they were issued. The only exception is when accessing Wilson as a slave. In this case, reads are performed in order and writes are performed in order, but Wilson may chose to violate order between reads and writes. The following is a block diagram of the SC:



System Controller Block Diagram

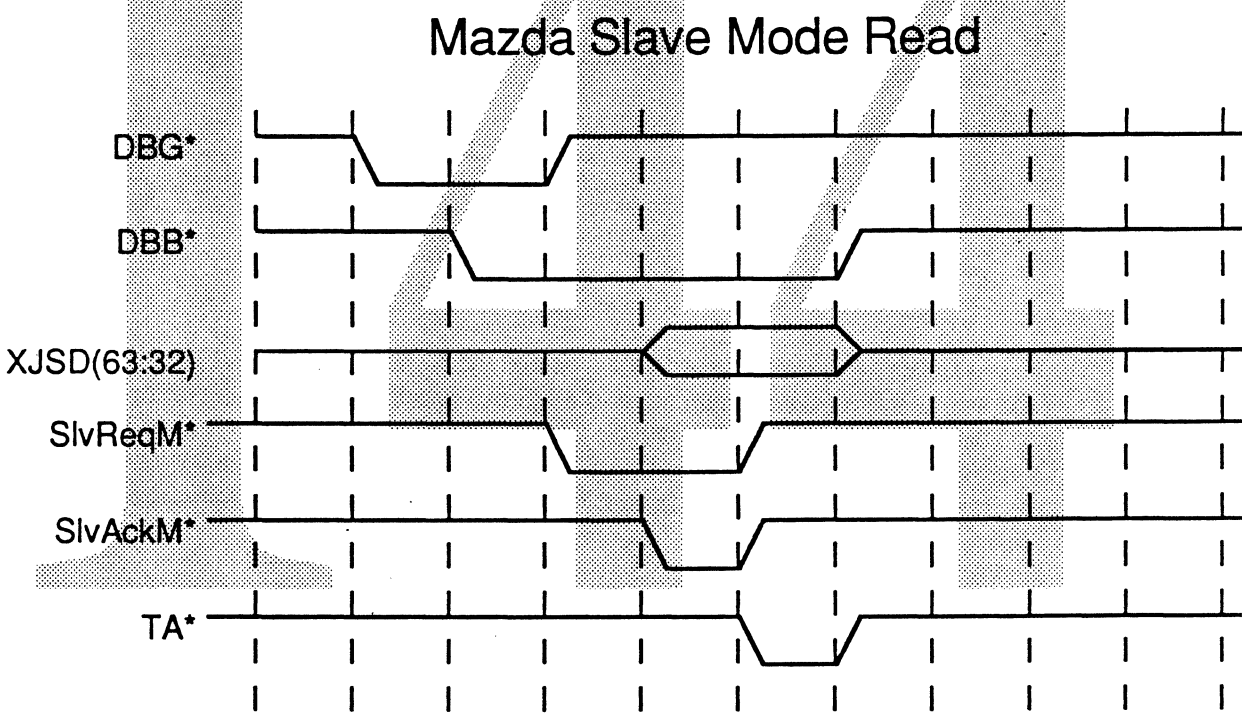
The XJSs and Mazda will access the SC as a slave as in the XJS Bus Specification. Mazda will only use the most significant 32 bits of the Data Bus (63:32). This means for Mazda a one beat transaction transfers 32 bits and a four beat transaction transfers 128 bits. The SC will use the address and transfer size information to determine the correct word alignment for the 32 bit access and cause the DPC to correct it to 64 bit alignment. In addition, Mazda will provide an interface to allow the access to the SC Status Registers. This interface will use an 8 bit data bus, IOBus(7:0), a 5 bit address bus, IOAdd(4:0), a write signal, IOWrite, and a chip select, IOSCSel, to access the SC Status Registers using protocol specified in the IO ERS. Since the address bus is byte addressed, registers which are multiple bytes require multiple accesses. The following registers are implemented.

Address	Register
00-02	Screen Address Register 17:0
03-05	Cursor Address Register 17:0
06-08	Screen Row Bytes 17:0
09-0B	Cursor Row Bytes 17:0
0D	Video refresh count 3:0
0E	Low Power Control 7:0
0F	Diagnostic Register 7:0

10-13	Error Address Register 31:0
14	Error Type Register 7:0
15	DRAM Type Register 7:0

These registers will be described in more detail later.

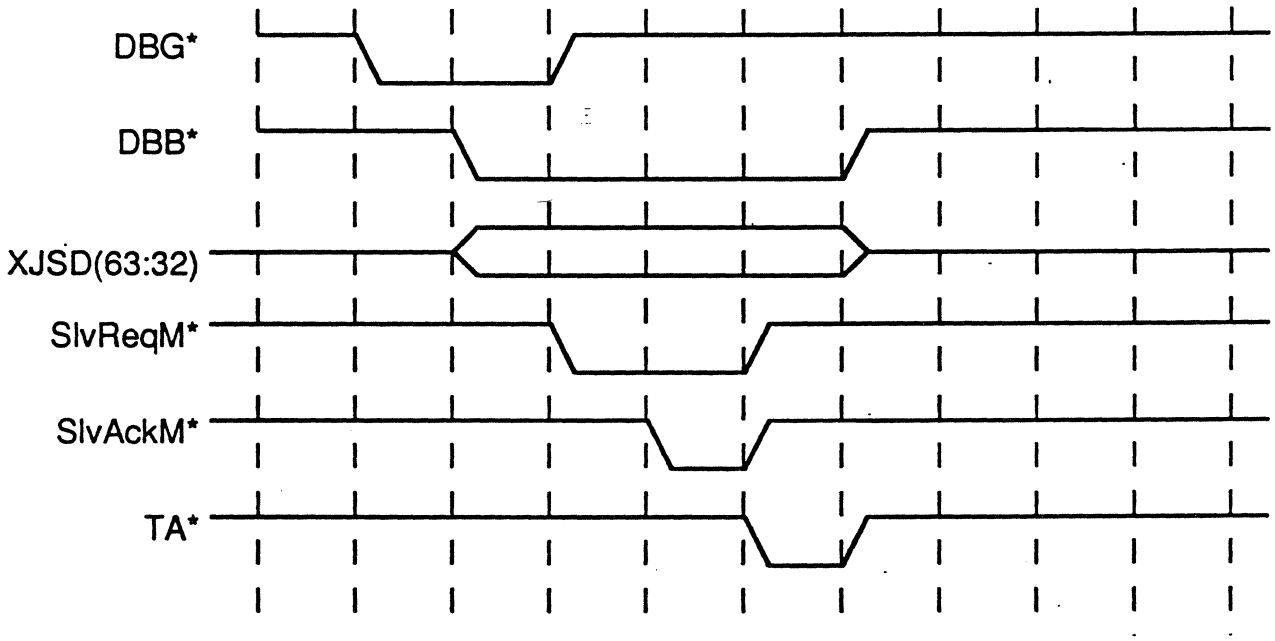
When Mazda is a slave, the SC uses the control lines SlvReqM* (Slave Request Mazda), and SlvAckM* (Slave Acknowledge Mazda), to coordinate data transfers. Since the SC acknowledges all addresses, the scoreboard for Mazda slave transactions must be functionally equivalent to the one in Mazda. This way, both the SC and Mazda implicitly know if an address can be accepted. They also know which transaction will be next. Mazda will only support single beat data transfers in this mode. A diagram of a slave mode read is shown below.



In this case data is being read from Mazda by an XJS. The same protocol works for other masters. When the data bus is free, the SC asserts SlvReqM*. This tells Mazda that it is free to drive read data onto the bus. When Mazda drives data it asserts SlvAckM* to indicate valid data and release of the internal address buffer associated with that request. Mazda must hold the read data until SlvReqM* is sampled inactive, at which point Mazda must turn off its drivers. The SC will deassert SlvReqM* one cycle before asserting TA*.

In the case of a write, data is written to Mazda by an XJS or another master. For a write, the SC asserts SlvReqM* to indicate that valid write data is on the bus. Mazda asserts SlvAckM* to indicate completion of data transfer and release of the internal address buffer associated with that request. The SC then drives TA* to complete the cycle. A diagram for a write is shown below.

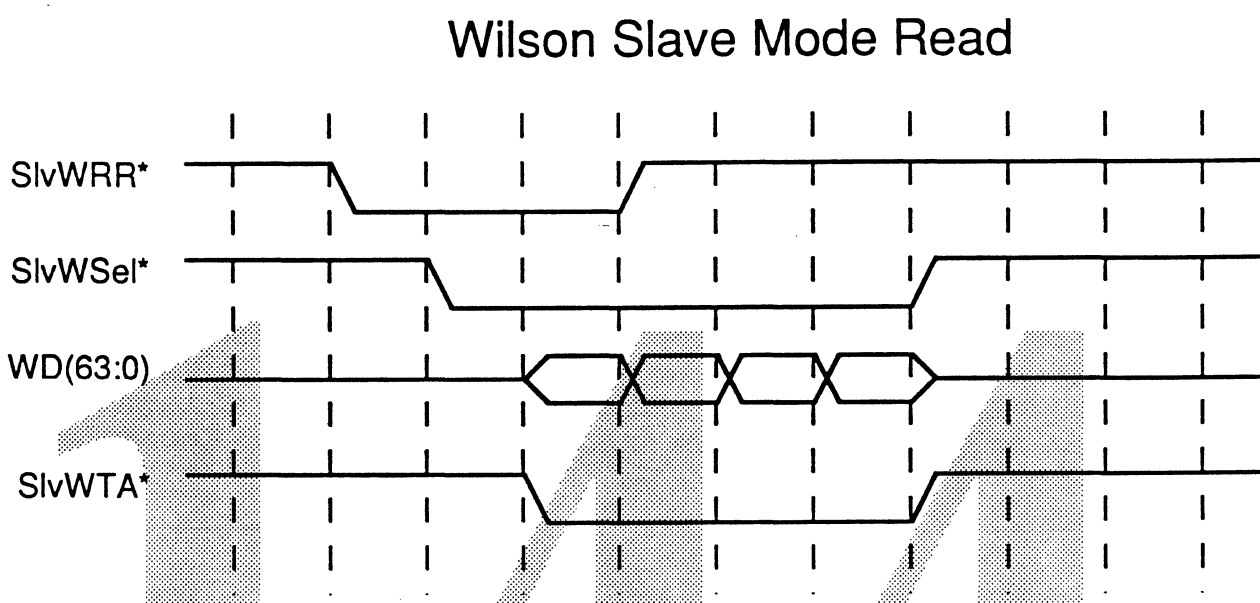
Mazda Slave Mode Write



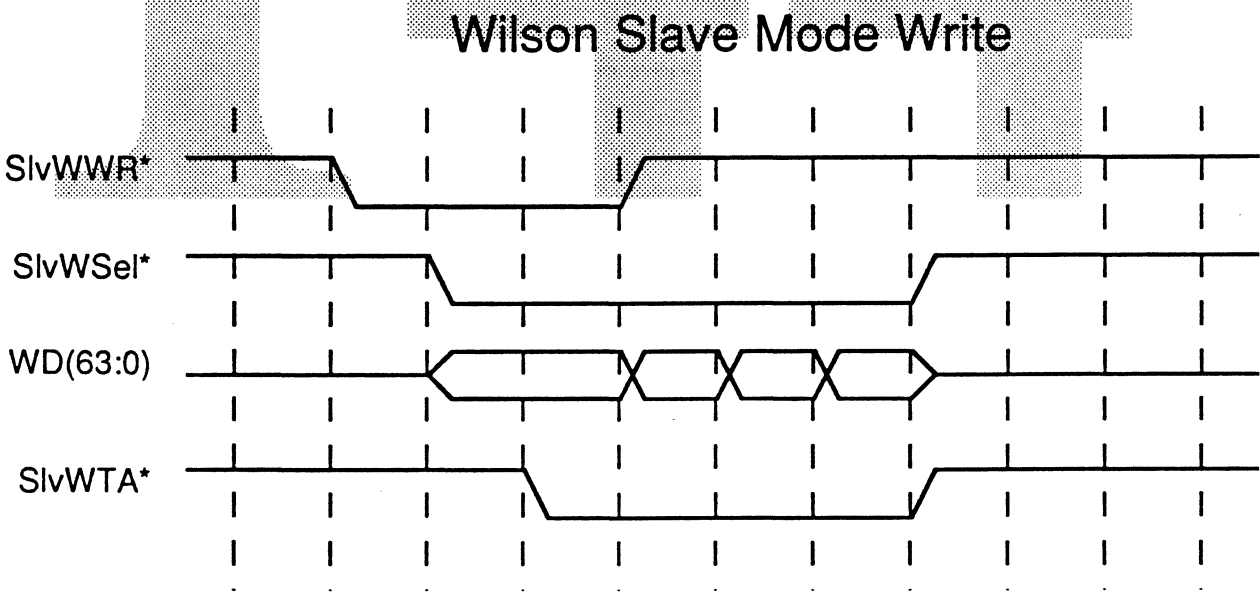
Wilson shares the address bus with the XJSs and the IO but has a separate data bus to the DPC. Wilson acts as a Bus Master according to the XJS Bus Specification, with the exception of using byte enable lines as described in the E & W ERS. Wilson also acts as a bus slave. When Wilson is acting as a bus slave, the SC is responsible for acknowledging addresses and coordinating data transfers through the DPC. To accomplish this the SC and Wilson use the additional handshake lines :

SlvWSel*	Slave Wilson Select
SlvWWR*	Slave Wilson Write Ready
SlvWRR*	Slave Wilson Read Ready
SlvWTA*	Slave Wilson Transfer Acknowledge
SlvWAR*	Slave Wilson Address Ready

The SC scoreboard for Wilson requests will be functionally equivalent to the one in WIlson. Both will know which the next read is and which the next write is. Wilson will acknowledge data transfers independent of the state of its internal address buffers. Therefore, Wilson will assert SlvWAR* to indicate there is an address buffer available. The SC uses SlvWAR* to enable address acknowledge for addresses in the Wilson address space. For a read, WIlson indicates that read data is available by activating the SlvWRR* line. When the data path is ready, the SC will assert SlvWSel*. This tells Wilson to drive read data. Wilson then drives data and SlvWTA* to indicate valid data. When the read transfer is complete, the SC deasserts SlvWSel. A diagram for a read is shown below.

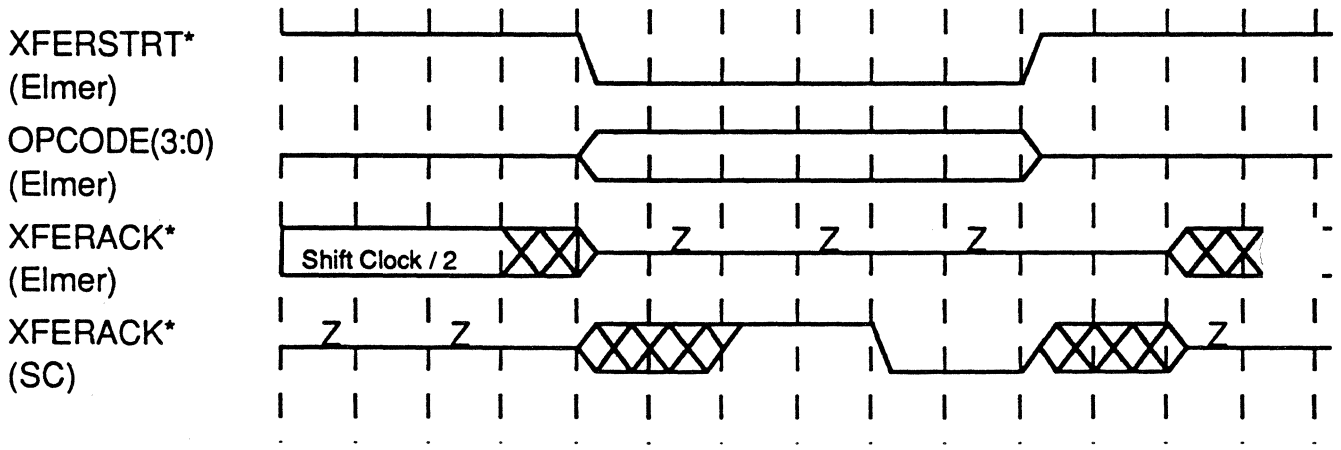


For a write, Wilson indicates that write buffers are available by asserting SlvWWR*. The SC will then drive write data and assert SlvWSel*. Wilson will capture the data and acknowledge the transfer by asserting SlvWSel*. When the transfer is completed, the SC will deassert SlvWSel. In the case of both SlvWRR* and SlvWWR* being active, the read will complete first. Race conditions will be handled by Wilson. A diagram of a write is shown below.



The SC provides an interface to the Video Backend Chip (Elmer). Elmer, contains the Video timing generation circuits. Elmer is responsible for initiating transfer cycles. After Elmer initiates a

transfer cycle, the SC will perform midline transfers to stay a half scan line ahead of the shift clocks until a new transfer is requested. To initiate a transfer cycle, Elmer issues a Transfer Cycle Start, XFERSTRT*, along with a Transfer Cycle Opcode, OPCODE(3:0). Transfer Cycle Acknowledge, XFERACK*, is a dual function signal controlled by XFERSTRT*. XFERACK* is normally an input to the SC, but becomes an output when XFERSTRT* is asserted. Therefore, when Elmer asserts XFERSTRT*, it must wait two cycles for XFERACK* to become valid. Elmer then samples XFERACK*. When the SC completes the specified transfer cycle, it will assert XFERACK*, and wait for Elmer to deassert XFERSTRT*. The SC XFERACK* will then become an input and Elmer may initiate another transfer. To generate the Screen address for the transfer cycle the SC provides the Screen address generator shown in diagram 2.1 which executes the opcodes in table 2.1. To perform the midline transfers, the SC must count shift clocks. ShiftClock/2 is sent to the SC as a second function of XFERACK*. When XFERSTRT* is deasserted, the bidirectional pin XFERACK* will be tristated in the SC, so that ShiftClock/2 can be driven by Elmer two internal clocks later. A timing diagram for these signals is shown below.



Refresh for the VRAMs will be done using CAS before RAS refresh cycles. A software programmable number of these will be done as an atomic transaction with each transfer cycle which performs a screen address transfer. There is also a special opcode to allow Elmer to initiate a refresh only operation. This is provided to allow Elmer to initiate refresh cycles during vertical blank.

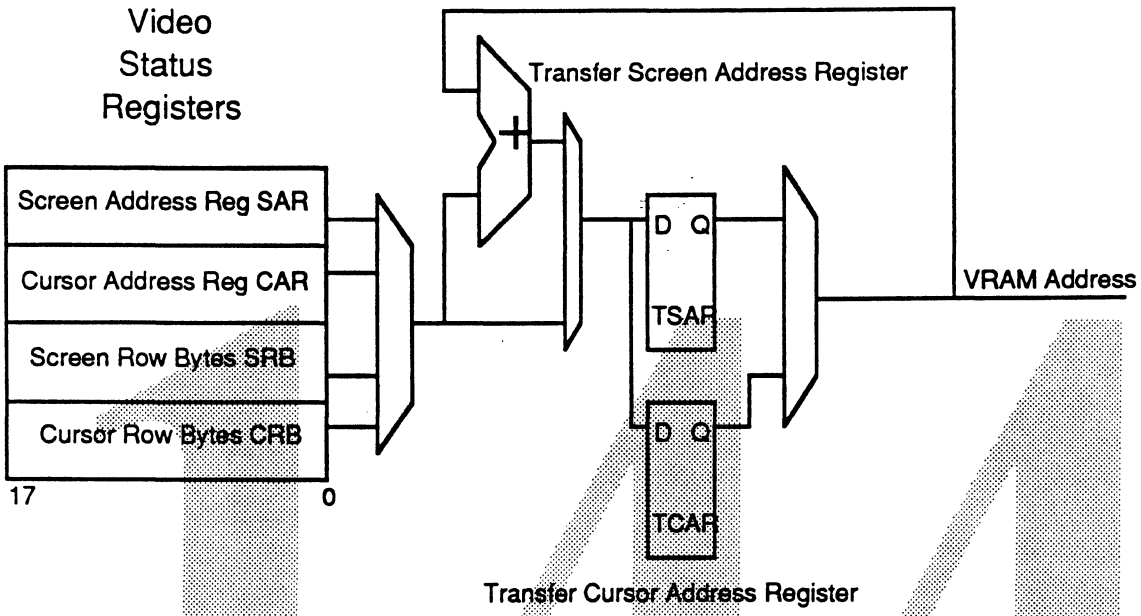


DIAGRAM 2.1

	Bank0	Bank 1		
Transfer	Transfer	Transfer	Next	Next
Opcode	Address	Address	TSAR	CSAR
0	SAR	SAR	SAR	TCAR
1	CAR	CAR	TSAR	CAR
2	TSAR	TSAR	TSAR	TCAR
3	TCAR	TCAR	TSAR	TCAR
4	TSAR+SRB	TSAR+SRB	TSAR+SRB	TCAR
5	TCAR+CRB	TCAR+CRB	TSAR	TCAR+CRB
6	TSAR	TSAR+SRB	TSAR+SRB	TCAR
7	TSAR+SRB	TSAR	TSAR+SRB	TCAR
8	TCAR	TCAR+CRB	TSAR	TCAR+CRB
9	TCAR+CRB	TCAR	TSAR	TCAR+CRB
10	SAR+SRB/2	SAR+SRB/2	SAR+SRB/2	TCAR
11	CAR+CRB/2	CAR+CRB/2	TSAR	TCAR+CRB/2
12	SPECIAL	REFRESH ONLY		
13	RESERVED			
14	RESERVED			
15	RESERVED			

TABLE 2.1

Diagram 2.1 shows the hardware mechanism for generating transfer cycle addresses. There are four, 18 bit registers, the Screen Address Register, the Cursor address register, the Screen Row Bytes Register and the Cursor Row Bytes Register. These are manipulated according to table 2.1 to produce transfer cycle addresses for Bank 0 and Bank 1 of VRAM.

The SC provides control lines for the two data path chips. A block diagram for the data path chip is shown in the DPC section. Queues are associated with each of the four possible destinations for each transaction. 21 control lines are used to control the DPC . Each queue has 5 control lines associated with it except the VRAM queue which has a bits per pixel BPP bit. There is also a parity error bit. For the VRAMS and the DRAMS the control bits are defined as follows:

1 Queue Output Enable
1 Queue Advance Enable
1 Queue Load Enable
1 Load Source Select
1 Load Hi/Lo Select
1 24 BPP mode (VRAM queue only)

For the CPU queue and the E & W queue the 5 bits are defined as follows:

1 Queue Output Enable Hi
1 Queue Output Enable Lo
1 Queue Advance Enable
2 Queue Load Select - these bits are encoded
    0 - Load from input 0
    1 - Load from input 1
    2 - Load from input 2
    3 - No Load

The SC is designed to control two 128 bit banks of DRAM and one 96/64 bit bank of VRAM. Each DRAM bank can contain from 4 to 64 megabytes, configured with 1, 4, or 16 megabit DRAM devices, while the VRAM bank is fixed at 1.5 megabytes. Optional parity checking is supported. Software will enable parity checking by a bit in the SC control registers if all SIMMs installed support parity. In the case of a parity error, the SC will signal the requesting master using the Transfer Error line. The address of the last error detected will be logged in Error Address Register.

The SC performs the following operations to VRAM: read, write, page mode read, page mode write, shift register reload, split shift register reload, refresh, and block write. The SC will support 100 nsec VRAMs. The cycles will be optimized to run at either 40MHz or 50MHz. The frequency can be selected by a bit in the VRAM Type register. There are three kinds of system accesses supported, 8 bits per pixel, 24 bits per pixel, and 24 bits per pixel with address translation. The address translation is a 7/8 reduction used for convolution mode video as described in the Frame buffer ERS. To distinguish the type of accesses, VRAM is mapped into three address spaces. These address spaces are defined below:

Address Space	hi address	lo address	total
VRAM Space	0x83FF FFFF	0x8000 0000	64 Mbytes
8 BPP	0x807F FFFF	0x8000 0000	8 MBytes
24 BPP	0x80FF FFFF	0x8080 0000	8 Mbytes
24 BBP AT	0x817F FFFF	0x8100 0000	8MByte

Shift register reloads occur during horizontal blank, and are initiated by Elmer. Split shift register reloads are initiated by the SC as needed to keep a half scan line ahead of the shift clocks.

For DRAM, the SC performs normal read and write operations, as well as page mode read, page mode write and CAS before RAS refresh. The SC will support 80 nsec, fast page mode DRAMs. The cycles will be optimized to run at either 40MHz or 50MHz. The mode can be selected by a bit in the DRAM Type register.

The Low Power register will allow software to turn off VRAM and DRAM refresh on a per bank basis. It will also allow software to disable VRAM transfer cycles.

Datapath

The DataPath Chip (DPC), provides the point to point interconnect for the systems data paths. There is no control logic or control state on the DPC. All control lines are driven by the SC. The DPC contains four data queues; the CPU Queue, the DRAM Queue, the VRAM Queue and the E & W Queue. These queues are associated with the destination of data they contain. The DPC is byte sliced. DPC1 is defined as containing XJS Bus bytes 1,3,5,7. DPC0 is defined as containing XJS Bus bytes 0,2,4,6. Parity is generated and checked for data as it passes through the DPC. Error signals are sent to the SC for reporting to requesting master. The DRAM Bus data has a two to one correspondence with the data on the XJS Bus. Even double words appear on DRAM Bus(127:64). Odd double words appear on DRAM Bus(63:0). For the VRAMs there is a different mapping for 8 BPP accesses and 24 BPP accesses. In 8 BPP mode 8 pixels are written at a time and in 24 BPP mode 4 pixels are written at a time. This is diagrammed below. The top three line show how data appears to the processor. The next two lines show the DPC byte slicing. The next two lines show how data in 8BPP appears to the DPC and how it is reorganized into the queue. 24 BPP data maintains its original format as shown on the next line. The next line shows the physical interconnect between the 128 bit VRAM Bus and the 96 bits of VRAM. Note that in DPC0, four bytes are unconnected. The final three lines simply show the pixel format at the VRAM serial port.

1 - Data Viewed at Processor

Half Cache Line

127:120	119:112	111:104	103:96	95:88	87:80	79:72	71:64	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
Word 0				Word 1				Word 2				Word 3			

8 BPP Processor Format

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15

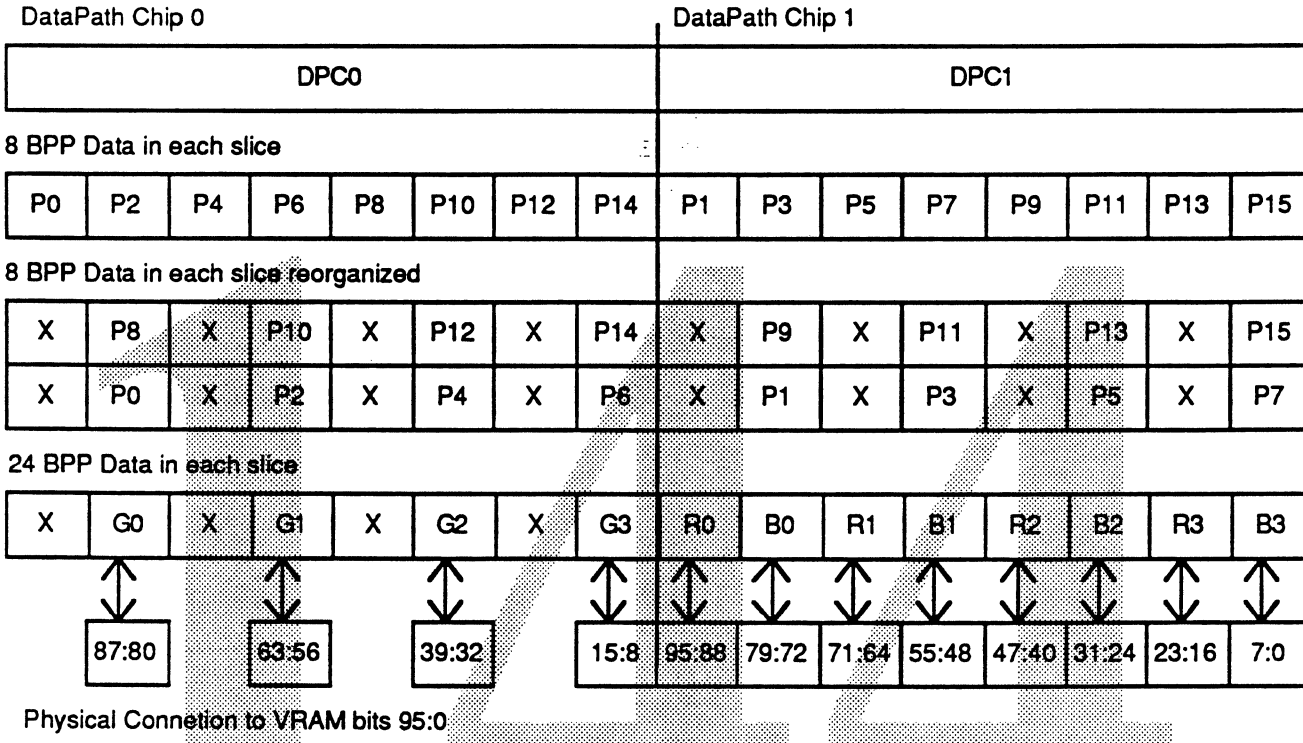
24 BPP Processor Format

X	R0	G0	B0	X	R1	G1	B1	X	R2	G2	B2	X	R3	G3	B3

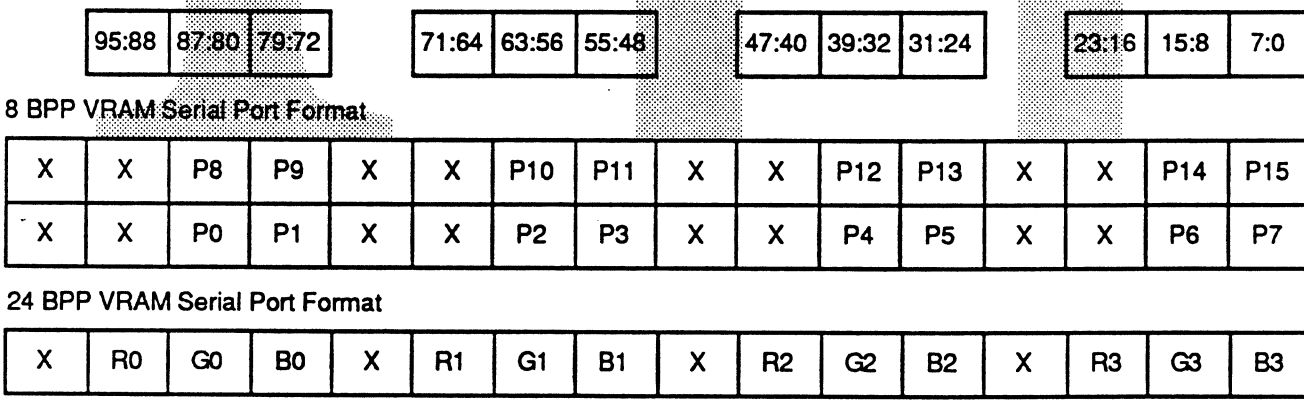
DataPath Chip Byte Slicing

DPC0	DPC1	DPC0	DPC1	DPC0	DPC1	DPC0	DPC1	DPC0	DPC1	DPC0	DPC1	DPC0	DPC1	DPC0	DPC1

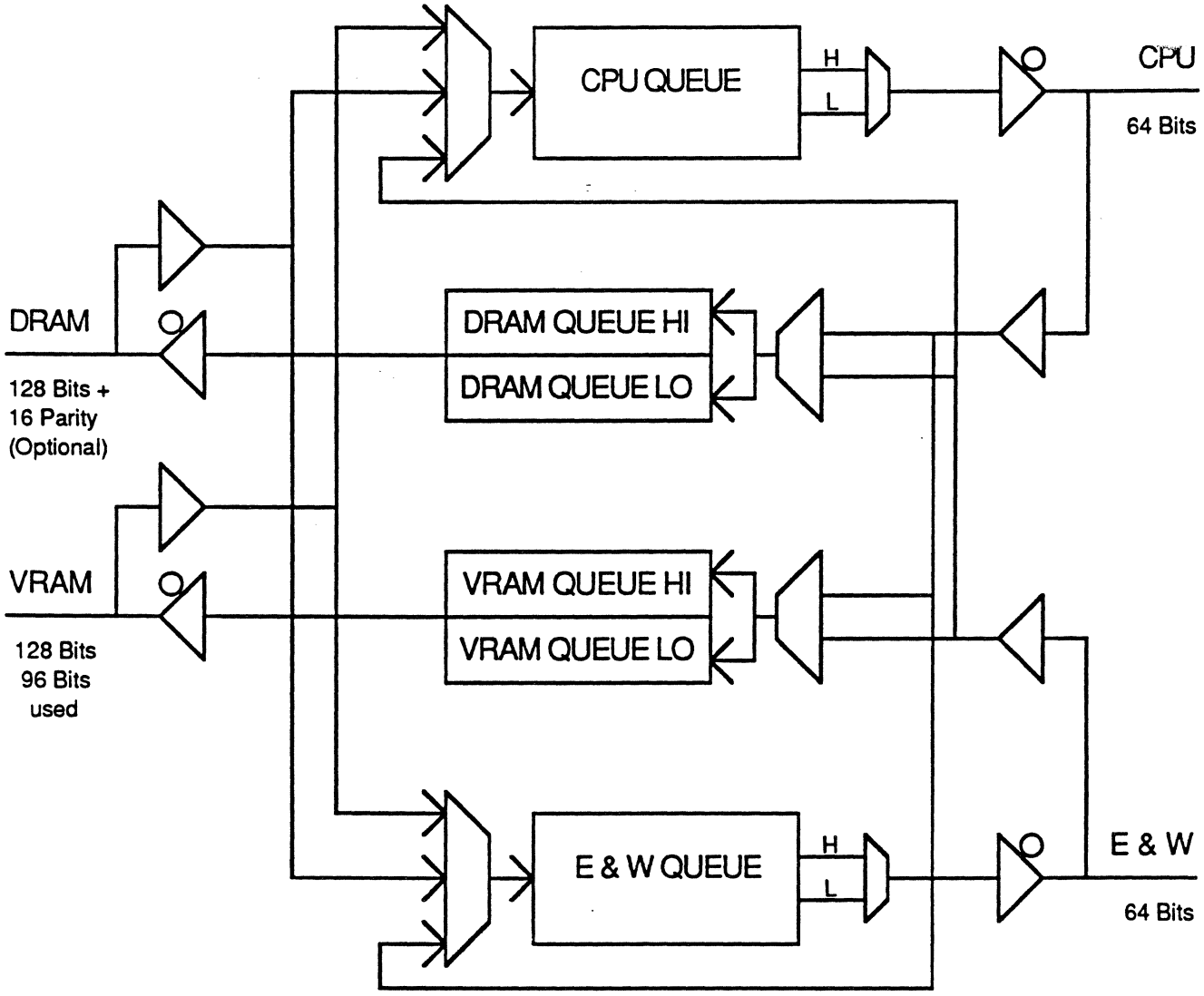
2 - Data Viewed By Datapath Chip Slice



3 - Data Viewed at VRAM Serial Port



Below is a block diagram of the DPC.

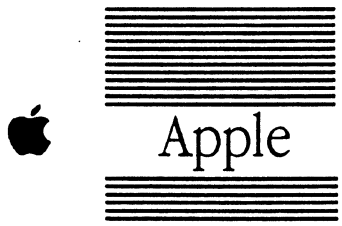


DATAPATH CHIP BLOCK DIAGRAM

The datapath is TBD.

System Clock

The system clock is either 40 MHz or 50 MHz, and is distributed using a Motorola PLL clock distribution chip. All chips on the board will receive the system clock with a jitter of +/- 1 ns.



Jaguar Graphics Overview

External Reference Specification
Special Projects

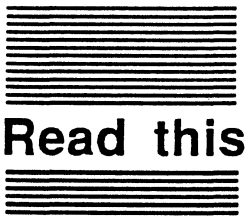
November 15, 1989

Return Comments to:
Juan Pineda
Phone: x4-9265
AppleLink: JUAN
MS: 60AA

144

Apple CONFIDENTIAL

Contents



Scope

This chapter of the Jaguar ERS introduces the basic building blocks that comprise the Jaguar graphics subsystem and gives an overall picture of how graphics operations work on Jaguar. The detailed specification of these blocks are given in the following three chapters of the ERS:

Jaguar Video Back End ERS

Jaguar Decompression ERS

Jaguar Expansion Interface & Wilson ERS

Definitions

NTSC	Broadcast standard for TV in the United States.
PAL	Broadcast standard for TV in Europe.
VRAM	Dual ported dynamic RAM that is used for graphics frame buffers.
CLUT	Color Lookup Table that is used to translate colors in 8 bit frame buffers or for gamma correction in 24 bit frame buffers.
DAC	Digital to analog converter. A chip that converts digital signals into analog signals that are required to drive monitors.
Back buffering	A technique that allows for the instantaneous update of a window. Images are rendered in main memory, and then transfered at high rate to the screen so that the entire image is updated between successive frames. Without back buffering, the process of rendering would be visible on the screen, and the effect of animation would be lost.
Z buffering	A technique for properly rendering overlapping objects when doing 3D rendering. The Z buffer is an array in system memory that contains a floating point number for each pixel in the window.

True color — True color refers to the ability of the graphics subsystem to display arbitrary colors on the screen. True color is often called 24 bit or 32 bit color because it generally requires 24 bits per pixel, eight bits for each color component (red, green and blue), and is represented in pixel maps in a 32 bit word. The alternative to true color is pseudo-color.

Pseudo-color — Pseudo color is a method by which colors are represented by an index into a color table that contains the full 24 bits required to represent the color on the screen. Since the index is typically 4 or 8 bits, this techniques reduces the number of bits required to store the image on the screen. The disadvantage is that the screen can display only 16 or 256 colors simultaneously.

Introduction: The importance of graphics on Jaguar

Jaguar will advance personal computing by incorporating new developments in processor, memory, VLSI and ASIC technology to raise the base level functionality and performance available for personal computing by at least an order of magnitude. This increase in computing power will enable the development of new personal applications that are not currently possible.

Essential to these new applications will be the ability to collect, sort, manipulate, synthesize and display images from a wide variety of sources. Photographic images will come from sources such as: FAX, high quality digitizers, still video, and recorded and live video. Synthesized images will come from 2D and 3D models of real and imaginary objects. Applications that perform simulation and use data visualization techniques will synthesize and animate 2D and 3D objects to more efficiently communicate with the user. Images will be scaled, enhanced, edited and otherwise modified in the course of their use. In addition, images will also be viewed in windows and will be mixed and overlaid with other images, such as pointers and menus and dialogue boxes.

The Jaguar graphics subsystem must provide the basic functionality and performance to enable these kinds of applications. The graphics subsystem must significantly increase the performance of existing rendering operations so that existing graphics techniques can be used to improve the quality of the user interface. In addition, new real time 3D rendering operations must be provided to extend the user interface paradigms beyond the current 2D space into 3D.

The Jaguar graphics subsystem must allow the display of full motion video images, and well as still images. These images must be displayed in a window with overlaid graphics. In addition, high resolution gray scale and color monitors that are properly corrected for high quality image and antialiased graphics display, must be provided. NTSC video output will be required to allow for storage and distribution of animated images on standard video tape formats.

Goals

The graphics functionality provided should be uniform across different Jaguar configurations so that applications can take full advantage of functionality without having to create different codes for different configurations. In practice, multiplicity of codes results in a combinatorial explosion of cases that becomes impractical to code.

The graphics subsystem should produce high quality images. The graphics subsystem is the window through which the users sees the Jaguar world, and it is important that the display provide a comfortable low stress display of this world. For example, high resolution fonts are easier to read and are less stressful to the user than coarser fonts.

The graphics subsystem should enable applications to display and synthesize images in real (interactive) time. Photographic images from stored or live sources should be displayed in full motion, and manipulated interactively. Applications should be able to synthesize and animate complex 2D and 3D images in real time.

Components

Graphics on Jaguar consists of several components, some of which are not so separable from other parts of the system. The fundamental components are:

Graphics monitor
Video back end
Frame buffer memory
System memory
E&W chip
Processor graphics instruction set/memory interface
Decompression accelerator
Expansion modules

The graphics monitor is the physical component through which the user views images displayed by the graphics system, and is usually the largest object that sits on the user's desk. Since the monitor is the primary interface that the user has to the system, quality is essential. Since the monitor sits on the users desk, size, weight and footprint are critical. The base Jaguar monitors will be 16" landscape color and 16" portrait monochrome.

The video back end converts the digital representation of pixels in the frame buffer into the analog signals necessary to drive the high res graphics monitor. In addition to converting the digital image into analog, the video back end must generate proper timing to drive monitors of different resolutions, and to drive TV video signals in NTSC and PAL format. The video back end is usually where gamma correction is performed to maintain proper color balance and saturation of displayed images.

Frame buffer memory maintains a digital representation of the image that is displayed on the monitor. The Jaguar frame buffer consists of 1.5Mbytes of dual ported VRAM. This frame buffer is large enough to drive the standard 16"Jaguar monitors.

System memory is used for many purposes related to graphics. System memory is used for back buffering of pixel maps in order to obtain clean image updates. In this technique, an image is rendered into system memory, and when completed transfered to the frame buffer. In this way, the image is updated to the screen in a single frame time. In addition, to back buffers, system memory will maintain alpha maps, Z buffers and display lists.

Window DMA provides for the fast transfer of images between the frame buffer and system memory or devices in expansion slots. The window DMA function will provide the capability to mask out pixels so that images can be overlaid, such as a pointer overlaid over a live video image. Window DMA is the primary mechanism for transfering back buffers to the frame buffer.

The Jaguar processor incorporates special instructions to enhance the performance of graphics operations. Special instructions are included to perform 3D graphics and alpha compositing.

The Jaguar decompression accelerator provides real time decompression of video encoded in standard formats such as CCITT P*64 or in Apple custom formats.

Expansion adaptors plugged into the expansion bus allow for extending the functionality of graphics beyond the point that is economically reasonable in the base configuration. For example, an expansion frame buffer adaptor will allow the use of a 21" monitor or provide a second display head on a system. Other expansion adaptors will provide video compression, video input, and rendering acceleration.

Configurations

Before getting down to the details of each of the individual components, the reasons for particular configurations should be discussed. Display size, resolution and number of colors (gray scale included) are the primary factors in the configuration that affect the graphics subsystem, and these are the issues that will be discussed in this section.

Because of the multimedia focus of the product, Jaguar must have a color monitor at introduction. Monochrome is important for high end publishing applications, but if we had to choose only one monitor, it would have to be color, not monochrome.

Screen size is critical to the usability of a Jaguar. Large size is important to allow for enough information to be placed on the screen simultaneously to be useful to the user. On the other hand, larger screens also cost more, take up more physical space on the desktop, and make the machine less movable a single person.

The current 13" monitor is clearly better than the 9" size of the MAC SE, but still feels small. Anyone that has worked with a 19" monitor will attest to the ease that it affords in being able to display a whole page, diagram or spreadsheet at one time. Unfortunately, the cost of such a monitor is quite high as well, too much to be a main stream monitor. The weight of a color 19" monitor is upwards of 80lbs, too much for the average person to handle safely.

We felt that there is a step in usability with the ability to display a full 8.5"x11.5" page of paper on the screen. It would desirable if this page could be displayed in both portrait and landscape orientations. Unfortunately, a square monitor would be about 18" diagonal, and would be almost as large and costly as a 19". A reasonable compromise would be a 16" rectangular format. Such a monitor would weigh 40-50 lbs.

Since we believe color would be primarily targeted at media manipulation applications, such as decompressed video and color photographic images, and since these are primarily landscape in format, it makes sense for the color monitor to have a landscape orientation. Similarly, since we expect monochrome to be targeted primarily at document preparation, and since most documents are taller than they are wide, it would make sense for monochrome to be in portrait orientation.

One overwhelming response that we've gotten on monochrome is that there is a need for significantly higher resolution than the current 72DPI. The drive here is to provide a display screen quality approaching that of paper. This is required for publishing and paperless office applications. Traditionally, monochrome has been the cheap graphics alternative, but this view is changing. The three Mac monochrome monitors available today all sell in about the same volume (3-4K/mo) yet two of them, the portrait and the 2 page, both sell for more than the 13" color (which sells at a rate of14-16K/mo).

The conclusion here is that a monochrome monitor must provide high quality and that quality is more important than cost. Since the cost of todays 15" portrait monochrome display is about the same cost as the color display, it would be reasonable to set the cost goal for the Jaguar monochrome display to be no more than the cost of the Jaguar color display. At this time we believe that 100DPI resolution is achievable at that cost, so the target monochrome resolution is 100DPI.

While it would be nice to go to higher than 72DPI on the Jaguar color monitor, this requires exceedingly fine shadow masks, probably around .21mm, and this technology is still quite expensive. While better resolution would be nicer, the drive to achieve the quality of paper copy that monochrome had does not apply to color.

While the standard Jaguar monitors will be the 16" monitors, the graphics system will be arbitrarily programmable to support smaller monitors, such as the current 13" Mac II color monitor.

Jaguar Graphics Display Configurations

Colors	8 bit gray	True color
Resolution	100DPI	72 DPI
Monitor size	16" portrait	16" landscape
Refresh rate	75Hz	75Hz
Memory configuration	~870x1150	~830x630
Total video memory	1MB	1.5MB

True color and gray scale

The standard Mac II provides 8 bit (minimum is actually 4 bit) color and one bit monochrome graphics. In 8 bit color, only 256 different colors may be displayed on the screen at one time. One bit monochrome allows the display of only two colors: white and black. While these limited color palettes were sufficient for Macintosh applications that display mostly symbolic images, this limited color space is inadequate for the reproduction of photographic images or for 3D synthesized images.

The standard Jaguar graphics system supports 24 bit **true color** and 8 bit **gray scale**. True color and 8 bit gray scale allow for the display of arbitrary colors on the screen, and allow the high quality reproduction of photographic or 3D images. The decision to provide this level of generality, rather than psedo-color or less than 8 bit gray scale was a difficult one as the cost of the additional bits is significant. While it would be desirable to choose the lower cost alternative, 8 bit color is problematical in the image rich environment that Jaguar will exist.

Photographic images or 3D synthesized images that are composed of arbitrary colors and are true color by nature. Techniques such as dithering that trade spatial resolution for color resolution , and allow for the display of true color images in 8 bit frame buffers would degrade real time graphics performance and image quality.

Even if the performance and quality of dithering were acceptable, there is the problem of allocating the the CLUT resource in an 8 bit system between the different applications. The difficulty here is that there are only 256 colors that are simultaneously displayable on the screen in an 8 bit system, and these must be shared among all the applications. Today, in cases when two applications require more than 256 colors in total, then entries in the CLUT are shared between the two applications, and the shared entry is set to the color required by the active application. This causes the other windows belonging to the other application to have it's colors changed into crazy colors, and it goes "technicolor".

This technicolor artifact is acceptable today's MAC because there is really only one active application, and the users attention is focused mainly on the windows belonging to that application. The typical Jaguar user would often have several applications running simultaneously, such as a video tutorial running simultaneously with a presentation graphics application. In such cases both applications must maintain valid colors, and this is something that cannot be guaranteed with only 8 bit color.

We did consider making true color an option, so that users that did not want to display images would not have to pay the additional cost. These users would loose the ability to display video and photorealistic synthesized images interactively. The primary concern here is that the interactive multimedia functionality of Jaguar would become a specialized function, since developers would be tempted to code software for the lowest common denominator. We feel that the interactive video and photorealistic images provide a strong enhancement to the power given to the user, and we feel that this ability is general purpose enough that it should be universal.

The cost consideration is much less of an issue when the declining cost of semiconductor memories is taken into account. While the cost of memory is declining, the amount of frame buffer memory required for graphics is not increasing at a corresponding rate. The reason is that amount of frame buffer memory is related to monitor technology, and that technology is not advancing at a comparable rate. Consequently, the overall cost of frame buffer memory will decrease in current and future generations of Jaguar.

Because of the declining cost premium for true color and 8 bit gray scale, and because we believe that it is important to make use of interactive video and photorealistic images in applications and become a natural part of the user experience, we have decided to make these features standard in Jaguar.

Monitor

The previous section discussed the merits of different monitor sizes and resolutions, so that will not be repeated here. The remaining issues with monitors relate to gamma correction and electronic contrast and brightness control.

Because Jaguar will be displaying photographic quality images, it is very important that the display subsystem produce consistent quality images. Contrast, brightness and gamma are the monitor attributes that affect this image quality. Because multiple applications will be displaying images simultaneously, optimization of these attributes for a specific application is not sufficient. Rather, the display subsystem must produce a calibrated reproduction, so that each application can count on the display quality, and compensate accordingly.

Gamma correction of the driving voltage to a monitor is required because spot brightness does not not respond linearly to driving voltage. This causes images to appear washed out, and can cause colors to shift in hue. It is possible to correct for this effect by supplying an inverse distortion (in the RAMDACs) of the driving voltage that cancels the distortion in the monitor. This is possible only if the amount of distortion presented by the monitor is known to the system.

Jaguar monitors should have a specified gamma so that the system can compensate for the gamma effect. Initial advice on this subject indicates that this calibration is not expensive. Supporting this statement is the fact that consumer TV's have a gamma calibrated to a value of 2.2.

Current monitors have brightness and contrast knobs that the user can adjust to compensate for room brightness and personal preference. Jaguar monitors will have software adjustable contrast and brightness controls. In addition, gamma correction will be software adjustable to compensate for room brightness and personal preference. Since these adjustments are soft, they may be saved and restored to calibrated levels for individual user preference.

Video back end

The video back end converts the digital image stored in the frame buffer to the analog signal required to drive the monitor. It is comprised of a CLUT/DAC (like the Brooktree BT458), a pixel clock oscillator and sync timing generation logic.

The Jaguar video back end will be programmable to support multiple screen sizes and resolutions as well as support both color and gray scale. Both the pixel clock and the sync timing will be programmable. Jaguar will support monochrome up to 100Mhz, and true color up to 57Mhz, which are the frequencies required to support the 16" Jaguar monitors.

The back end will also be able to drive a TV or VCR with composite or S-VHS video in NTSC and PAL formats. When driving NTSC and PAL, the Apple proprietary convolution digital filter will be enabled to reduce the flicker inherent in interlaced display formats.

Gamma correction will be performed by loading the lookup tables in the RAMDAC with the compensating function for the gamma of the monitor.

The video back end consists of three VLSI chips. The ELMER gate array contains the necessary data paths to multiplex the 96 VRAM data outputs down to the narrower RAMDAC inputs, has circuitry to support convolution, and contains the sync generation circuit. The CLUT/DAC chip converts the digital pixel data from ELMER into the analog RGB signals required to drive the highres graphics monitor. The DENC (digital encoder) chip is similar to the CLUT/DAC chip, except that it generates the output to drive the composite and SVHS NTSC/PAL outputs.

In addition to these three VLSI chips, a programmable phase lock loop chip is used to generate the the pixel clock.

Frame buffer

Frame buffer memory contains the digital representation of the screen display. Frame buffer memory is mapped into the CPU address space so that applications can render directly into it. This memory is composed of 12 128Kx8 VRAMs for a total frame buffer size of 1.5Mbytes. This amount of memory is sufficient to support monitors up to the size and resolution of the standard 16" Jaguar monitors.

The standard 16" 72DPI color monitor displays .5Mpixels, so the 1.5Mbytes of frame buffer memory allows for 24 bits per pixel which allows true color to be displayed.

In 8 bit gray scale mode, only 1Mbyte of the frame buffer RAM is usable, rather than the full 1.5Mbytes. The reason is that the memory is that the 1.5Mbytes is actually distributed in a 2Mbyte address space, with one byte missing every 4th byte, since this is the format that is required for true color. In monochrome mode, we drop an additional byte out of every 4 to get 2 usable bytes every 4, and these are multiplexed down to a 1Mbyte address space. To do otherwise would require pixel twisting logic that would require additional interconnection between the two memory crossbar data path chips, and would introduce much complexity in the addressing logic. Since the standard 16" 100DPI gray scale monitor, 1Mbyte is sufficient to drive this monitor.

The frame buffer also has special addressing to support the convolution feature on interlaced video output.

Wilson

Because the display of interactive synthesized and digitized images is so important to the multimedia aspect of Jaguar, special hardware is required and has been included in Jaguar to facilitate the dynamic compositing of moving images from many sources onto the screen. Synthesized images will be rendered by the processor into system memory or by accelerators added to the expansion bus. Digitized images will be expanded from compressed data on disks or from ISDN phone lines or come directly from video digitizers on the expansion bus. These changing images must be composited onto one or more screens in real time with minimal artifacts.

The E&W (Expansion interface and Wilson) chip is the hardware that supports screen compositing on Jaguar. E&W supports the transfer of streams of 8, 16 and 32 bit pixels between devices, such as expansion adaptors, system memory or frame buffers. Transfers to memory or to the mother board frame buffer, map these streams into rectangular blocks by keeping track of rowbyte information when incrementing the memory address. In addition to transfering rectangular blocks, E&W supports the masking of a pixel transfer by a mask plane. This mask plane allows for overlays over dynamic data, such as live video, and for non rectangular or translucent windows.

E&W must operate at high speeds, and must have the ability to synchronize transfers to the frame buffer with the beam position. In order to animate windows and objects on the screen, the parts of the screen that change must be completely updated between successive screen refreshes. Unless the change is complete, the user will see tears (half of one frame with half of another) or incomplete intermediate images, and this will destroy the animation effect. Synchronization of transfers to the frame buffer require synchronization with the beam position so that updates to the screen always complete in a single frame.

The case of compositing to the screen from an image rendered into main memory is commonly known as back buffering. While it will be possible for applications to render directly to the frame buffer, it is expected that most applications will not want the resulting artifacts, and will be back buffered. Consequently, the assumption of Jaguar and E&W is that virtually all windows will be back buffered.

In addition to compositing windows, E&W will be used to clear or initialize windows. An E&W operation could be used to clear a back buffer or Z buffer to a constant value, or to a preset value. This could be used to paint the background in an animation. All of these E&W operations would operate in parallel with the CPU, leaving the CPU free to compute the next image.

The window DMA hardware also has an 256x24 bit CLUT that can be used for color translation on window updates to the frame buffer. This allows 8 bit back buffers to be used for applications that don't require true color, thus reducing the size of these back buffers by a factor of four. Since the color translation is performed on the way into the frame buffer, this CLUT does not have the same problems that a shared CLUT in the video back end would have.

Processor and graphics instructions

The Jaguar processor has a performance of 30-40 times the speed of a MAC IICX in integer benchmarks, and up to 200 times in floating point benchmarks. In addition, unlike MAC II frame buffers, which are limited by the performance of NuBus, the Jaguar frame buffer will have performance equal to that of main memory. This level of raw performance will enhance the performance of existing graphics operations, like those implemented in Quickdraw, by at least an order of magnitude over IICX and even 68040 implementations.

This increased level of performance, combined with Window DMA, will enable better operation of existing 2D graphics techniques. Higher quality animation will be possible, including the moving display of stored color images. Using processor or window DMA operations, BLT rates greater than 8 Mpix are possible, which may allow for the smooth scrolling of windows.

In addition to improving the performance of existing 2D operations, Jaguar needs to achieve good performance in a number of new graphics areas. In order to allow extensive use of antialiasing to improve the quality detailed graphics, such as text and cursors, it is important that Jaguar perform alpha compositing efficiently. In order to allow the interactive rendering of 3D graphics images, Jaguar should perform interpolation of true color shaded polygons, and Z buffer compare operation quickly. More general pixel arithmetic, with saturation is important to allow for imaging algorithms, such as scaling and rotation to operate quickly. Finally, because of the importance of animation, high speed transfer of images from memory to frame buffer is important to allow back buffer techniques to be used.

In order to increase the performance of these operations, the processor instruction set has been enhanced to provide special instructions for some of these operations. In some cases, special instructions were not cost effective because they could not provide much additional performance beyond the performance provided by the general purpose dual ALU architecture. Specifically, instructions have been added for interpolation, Z buffering and compositing.

Pixel arithmetic instructions have been added that perform multiple adds in a single instruction. These instructions allow a true color alpha compositing loop to operate at approximately 10 machine cycles per pixel, which yields a performance of approximately 4 Mpix. As a rough idea of what this performance would mean (although this is not really a meaningful operation), the entire 16" screen could be composited with another source at a rate of 8 times per second.

The pixel arithmetic instructions can also be used for interpolation of pixel values. In addition, a special Z buffer compare instruction has been added that allows for multiple Z values to be compared in a single cycle. These special graphics instructions, combined with the dual instruction execution pipe, make it is possible to render one true color (32 bit interpolations), Z buffered (using 32 bit floating point) pixel approximately every 8-10 machine cycles. Estimating 20 machine cycles of overhead per scan line, and 100 machine cycles of setup, a "typical" 100 pixel triangle (15 scan lines high) would render in 1200-1400 machine cycles, or 30-35 micro seconds. This yields a performance of ~30K triangles/second. Much of this time is taken by memory overhead. We are still tuning this operation, and it may be possible to achieve even greater performance.

While not specifically part of the graphics instruction set, the floating point instructions provide exceptional performance for graphics operations. A 4x4 matrix multiply used for 3D transforms, can be performed in 21 machine cycles, which results in a performance of 2 Million transforms per second. Based on published work of the number of machine cycles taken by machines with similar architectures, we could expect the traditional display list traversal, transform, light model, and setup for a single 3D triangle to take approximately 600-800 machine cycles, or 15-20us.

Combining transform and render performance estimates above, we could estimate Jaguar performance for the traditional 3D transform and render operation, at approximately 20K triangles/second. This level of performance is comparable to the level of performance available in todays 3D graphics workstations.

This rough estimate follows the traditional workstation model for graphics, and is useful for comparing Jaguar to traditional implementations. Newer architectures for graphics will represent surfaces directly, rather than specifying individual triangles, and may be able to reduce the transform/light model time. Even so, the limit will be the rendering time, which is still limited by the inner loop performance of 4 Mpix (10 machine cycles) for Z buffered operations. Non-Z buffered operations would operate faster, and could perhaps double the performance of the inner loop.

Because the Jaguar architecture uses the CPU to perform the graphics operations, and because there is much parallelism inherent in graphics algorithms, Jaguar graphics will be able to take advantage of multiple processors to increase performance. The graphics algorithms can either be divided functionally: using one processor for rendering, and one for transforms, or divided uniformly: using processors to transform and render complete triangles in parallel.

A major difference between the Jaguar CPU and traditional CPUs is the relative performance of floating point. In traditional machines floating point operations are usually much slower than integer operations, this is not the case with Jaguar. The Jaguar CPU can perform a floating point add and multiply in a single machine cycle, which is just as fast as what is possible with integer operations. In general, it can be argued that the fundamental limits of ALU designs dictate that for equal precision, floating point multiplies will always cost less and be a little faster than integer multiplies, while integer adds will be a little faster and cost less than floating point adders. We expect that all Jaguar CPUs in the future will have floating point performance approximately equal to their integer performance.

The relative performance of floating point and integer operations on Jaguar should have a profound affect on Jaguar applications. In general, applications written for Jaguar should use 32 bit floating point instead of 32 bit integer for cases where the exact behavior of integer operations are not important. An example where we have already followed this philosophy is in the choice of 32 floating point for Z buffer values. In this case, the use of floating point eliminates many of the range and overflow issues that must usually be dealt with when using integer Z buffers. Similarly, it would be reasonable to maintain 2D coordinate precision in floating on a Jaguar. It may even make sense to perform some interpolations, such as triangle boundaries in floating point.

Video Decompression

Jaguar is targeted at display and manipulation of visual media. Stored video images of TV news events, training films, demos, even family films will be stored on local disks or file servers and edited into presentations, documents or other films. Teleconferencing images will come over the integral ISDN phone connection that will enable the user to more efficiently communicate with others and realistically enable telecommuting applications.

The manipulation and storage of video requires compression and decompression of video because the data rates of uncompressed video are just to great to allow for storage and transmission with available technology. The raw bandwidth required for NTSC video is about 40Mbytes/second, while compressed video requires 300 times less, or about 150Kbytes/second. At 150Kbytes/sec, one hour of video can be stored in about 500Mbytes, or one optical disk. That data rate is also achievable over standard LANs.

Video teleconferenceing over 64Kbit ISDN channels requires a high degree of compression to get moving images down to 64Kbits/second

There will be international standards for both teleconferencing and high quality video compression. Currently CCITT has developed a standard for teleconferencing known as P*64. This standard has gained wide support and is based on DCT compression techniques. ISO is working on a standard for video compression known as MPEG, but this standard is still quite early in it's evolution. MPEG will be based on DCT techniques as well, and will also be upward compatible with P*64.

Because we expect the important standards to be P*64 and MPEG, and because the quality of DCT based algorithms are good for these applications, Jaguar will support decompression of P*64 and MPEG formats, and also be generally programmable to support other formats that are DCT based.

While it is important that Jaguar support as many of the industry standard formats as possible, in order to be able to import and export data to other environments, it is also important that Jaguar promote a single standard to be used by the typical application.

While Jaguar will provide decompression as standard, compression will be optional. Real time compression is a much harder problem than decompression and will cost correspondingly more. In addition, real time compression is also not generally useful without a video digitizer. The cost of video input combined with compression is much greater than that of decompression, and would add much additional cost to the system. An expansion card will be provided that provides compression and video input.

There are three choices here: 1) have no standard support for video compression or decompression, 2) support only decompression, 3) support video input, compression and decompression. Jaguar has chosen option two. The argument here is that decompression is more important than compression, because it is required to view video documents, and it is most important in a network is to make all information in the network readable by all users. If video documents were readable only if one had a special card in one's system, then video documents would not be generally distributed. While compression is required for recording video, authoring of video material is still possible with only decompression if all the components have already been digitized; compression only buys the user the ability to digitize and record video in real time.

The argument also says that if one wants to digitize video, one would not want to do that in one's office, but rather one would want to have a space allocated with proper lighting conditions, space and a backdrop. This resource could be used much the same way that shared printers are used today.

The only really strong argument for providing video compression and digitizing is for video teleconferencing. In that case, much like a telephone, it would be desirable to have the video input at the users desk. On the other hand, one could also have a conference room set up for teleconferencing. The conference room could have controlled lighting and sound conditions that would solve some of the problems with desktop teleconferencing. We cannot justify the cost of video input an compression in every Jaguar on this application at this time.

The problem of decompression on a personal computer in the Jaguar price range is a difficult one, and to our knowledge, it has not been solved for DCT based algorithms. At the time of the writing of this ERS, not all of the problems have been solved, and there is a risk that we may not achieve all of our goals.

Typical graphics scenario

This section presents a typical graphics scenario and shows the flow through the graphics subsystems. The scenario shows two applications running simultaneously: a 3D graphics application and a live video application with data coming in over a local area network.

The 3D application takes it's data from a data base on the local disk. It transforms and renders the object into the image buffer, and uses the intermediate Z buffer on each frame that it renders to perform hidden surface removal.

The application takes the objects in the data base and alternates between the geometry and lighting operations and the rendering operations for each triangle. The geometry and lighting operations are floating point intensive and produce the coordinates of triangles that are to be rendered into the frame buffer. Each coordinate consists of six components: the X and Y position in screen coordinates, the Z value and the three color components: R, G and B.

The rendering operation takes these coordinates and draws a linearly interpolated triangle into the frame buffer, performing a Z buffer compare and update on each interpolated pixel. Rendering takes advantage of the graphics instruction set incorporated into the Jaguar processor. The rendering operation is quite memory intensive, and can consume virtually all of the bandwidth of the machine. For typical 100 pixel triangles and simple single light source lighting computations, the geometry and rendering operations take approximately the same time.

When an entire frame is finally completed, the 3D application requests Animation toolbox to update the screen with the new frame. The process then goes on to work on the next frame. The image buffer probably has to be double buffered, since it will take a while for the image to be transfered to the screen, and it would be better if the 3D application did not have to wait for this to occur before continuing on to the next frame. For the sake of this example, the 3D application is generating frames at a rate of 10 frames/second.

Meanwhile, the decompression application is operating on a stream of data coming in over Ethernet. Perhaps the data is a tutorial on how to use the 3D application, or perhaps it is a teleconference to another user.

The raw data comes to the decompression code via the standard I/O system in I/O buffers. The decompression code strips off the coded block data from the header information, and instructs the decompression hardware accelerators to perform the appropriate operations on the block data to generate the next frame in the sequence.

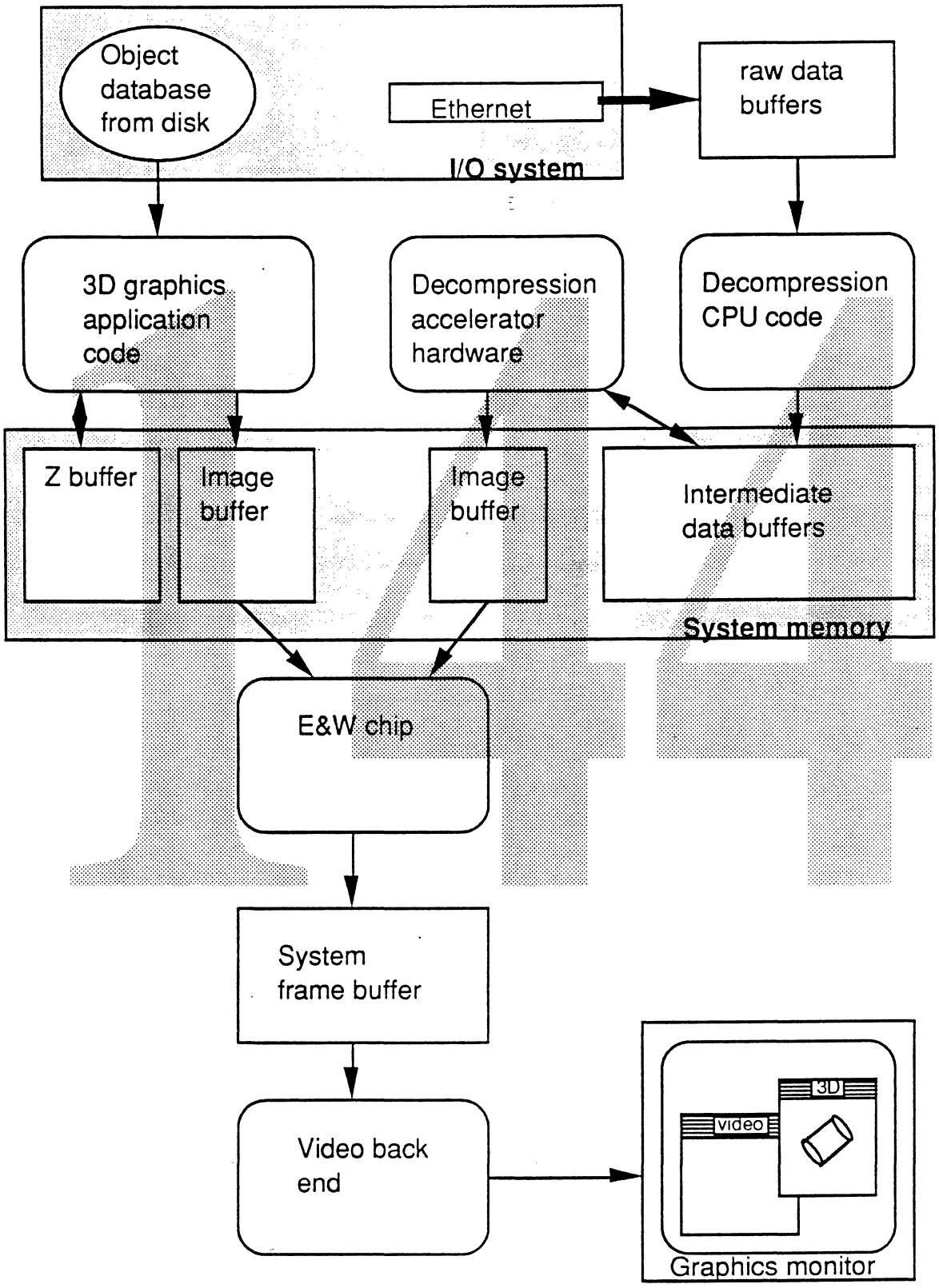
Once the image is completed, the decompression application requests the Animation toolbox to update the screen with the new image. Again the image buffer will probably have to be double buffered to allow the application to continue on without waiting for the buffer to be transfered. A typical decompression format would generate an image 360x288 pixels at a rate of 30 frames/second.

The Animation toolbox is not shown in the picture, since it does not actually touch any of the data. The Animation toolbox talks to the E&W manager to schedule the requested screen updates. The E&W manager schedules the events and requests the Wilson Driver to perform the operation. The Wilson driver finally sets up E&W to perform the actual data transfer. The data is transfered to the frame buffer. The mask plane is used in performing the transfer to the screen to properly clip the window if it is partially obscured.

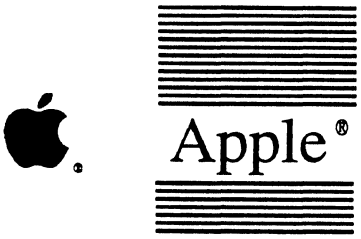
Each of the updates to the screen were scheduled so that they completed between successive screen refreshes. This is important to insure that the frames do not "tear" as they are updated.

The image in the frame buffer is sent to the screen 75 times per second by the video back end. The frame buffer is composed of dual ported video RAMs, and the serial port goes to the video back end. Frame buffer data is Scanned out serially from left to right and top to bottom. The pixels are passed through the CLUT that has been programmed to compensate for the gamma distortion of the monitor. The output of the CLUT goes through D/A converters that convert the pixels to analog voltages that are required to drive the Jaguar graphics monitor.

The final result is two animated images, one in a partially obscured window. The images have proper saturation and color balance for being properly gamma corrected.



Typical graphics scenario data flow



Jaguar Video Back End

External Reference Specification
Special Projects

Nov. 14, 1989

Return Comments to:
Bill Dawson
Phone: x4-0640
AppleLink: DAWSON.B
MS: 60-D

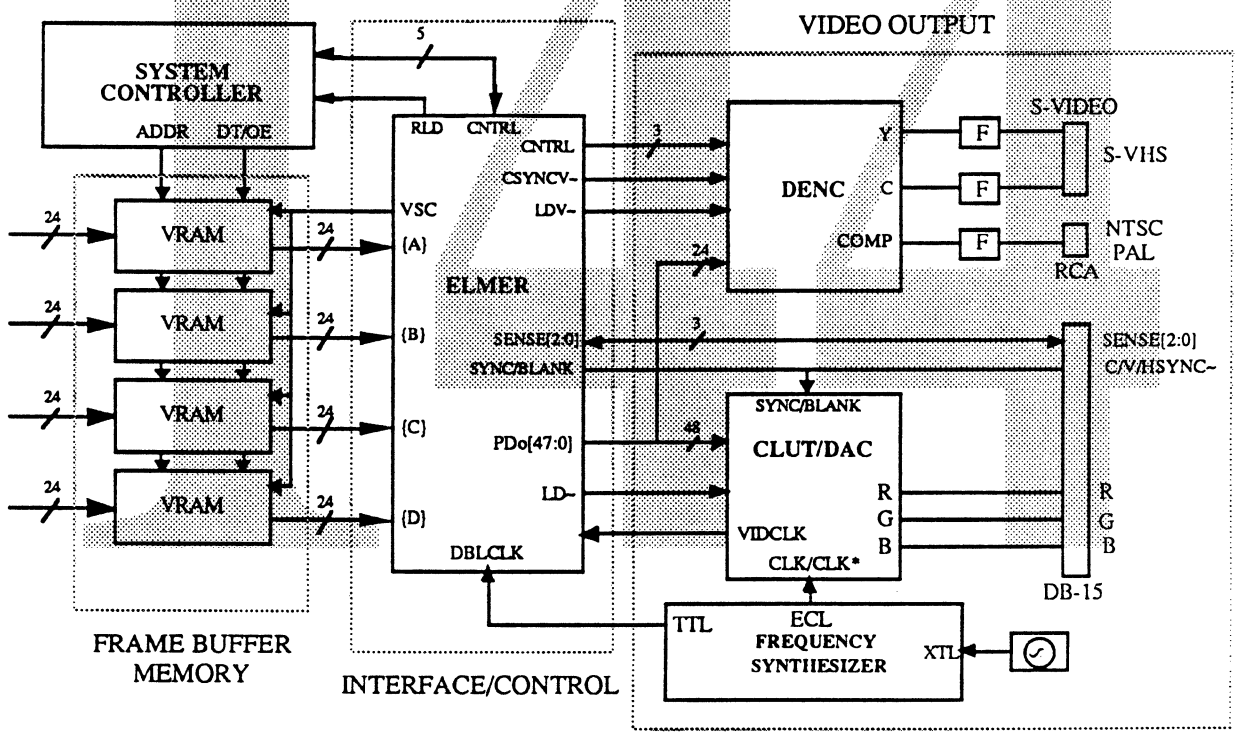
Apple CONFIDENTIAL

Contents

Section 1

Introduction

This external reference specification covers the hardware support on the motherboard for the entire video backend. As illustrated in the block diagram below, it covers the frame buffer organization, the interface between the frame buffer and the video output circuitry, and the component RGB and encoded NTSC/PAL video output. The paper contains three main sections: 1) the frame buffer organization; 2) the interface, video timing generation and control of the frame buffer; 3) the analog circuitry to generate the component and encoded video outputs.



The frame buffer organization section provides a brief description of how the different pixel formats are mapped into the frame buffer RAM. This defines the frame buffer RAM as seen from the processor and the video refresh hardware. Related document is the Memory ERS.

The interface/controller section describes the proposed functionality of the frame buffer glue chip, Elmer. Elmer serves three main purposes: 1) to multiplex, buffer and reorganize the four 24-bit, Video RAM serial ports into a high speed CLUT/DAC interface; 2) to generate the video timing to support a wide range of monitors and to control the Video RAM serial ports; 3) to enhance the JAG frame buffer by adding functionality as possible, such as 24-bit, true-color, convolution filtering and limited hardware cursor support.

The video output section details the CLUT/DAC and digital video encoder (DENC) subsystems which convert the digital pixel data to the component RGB and encoded NTSC/PAL video outputs, respectively. Finally, a short section is presented on the programmable pixel clock generation and critical timing parameters.

Section 2

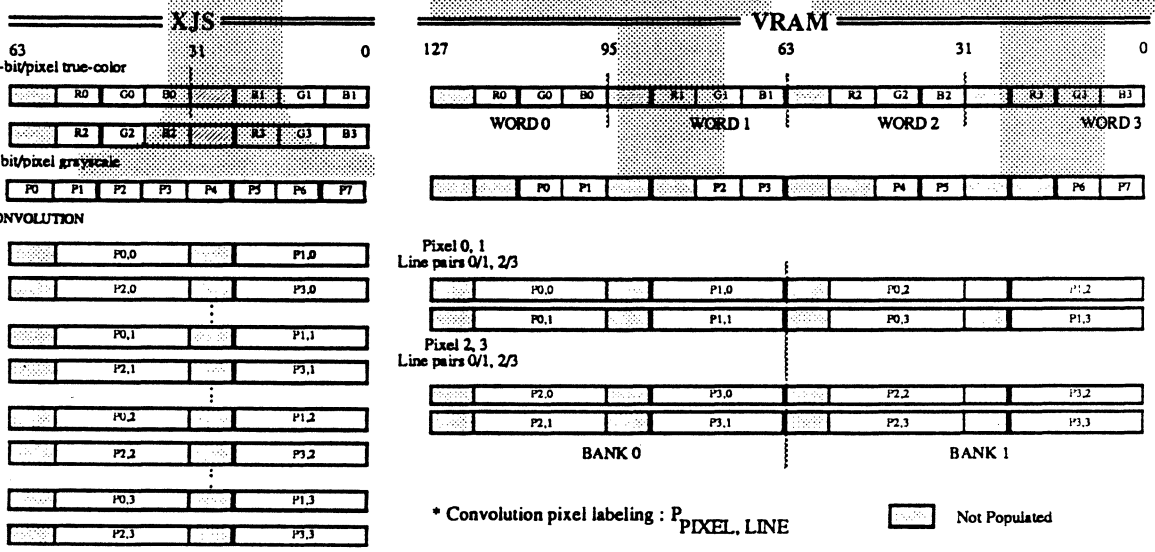
Frame Buffer Organization

Frame Buffer Addressing

The frame buffer is fixed at 1.5 MBy of Video RAM, and is reconfigurable depending on the number of bits per pixel. When it is used with 8-bits per pixel, the memory is seen as 1 MBy of RAM with a 64-bit bus, or 8 pixels wide. At 24-bits per pixel, it is seen as a 128-bit wide memory with every fourth byte missing, or 4 pixels wide.

To support 24-bit per pixel convolution filtering, the 128-bit wide memory is partitioned into two 64-bit wide banks. Each bank is seen as a 64-bit wide memory with every fourth byte missing, or 2 pixels wide. Successive scanlines are interleaved on a dual pixel basis to form line pairs. Successive line pairs then alternate between the banks, such that line pairs 0/1, 4/5, 8/9, ... are stored in Bank 0 and line pairs 2/3, 6/7, 10/11, .. are stored in Bank 1. This allows pixels from four adjacent scanlines to be clocked out of the video RAM serial ports in quick succession, two 96-bit accesses.

The diagram below shows the byte lane ordering of the different pixel formats as seen from the processor bus and the Video RAM serial ports.



A 64 MBy address space is allocated for the frame buffer in the system memory map. The 64 MBy address space is further divided into 8 MBy address spaces which are redundantly mapped into the VRAM for the different modes of operation: 8-bit/pixel, 24-bit/pixel and 24-bit/pixel convolved. 8-bit/pixel convolved is not supported because of the added complexity and degradation in bandwidth.

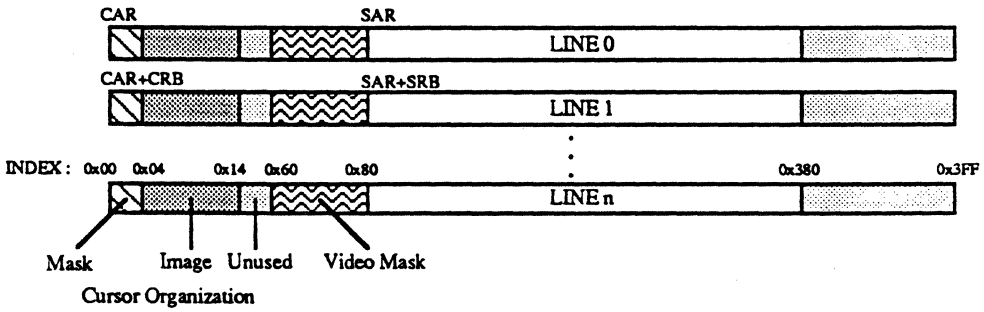
This frame buffer organization can support the following pixel formats and monitors.

24-bit/pixel, True-Color		
NTSC	640 x 480 Convolved, full-screen	
PAL	768 x 576 Convolved, full-screen	
13" Landscape	640 x 480	
16" Landscape	830 x 630	(preliminary)
8-bit/pixel, Grayscale		
13" Landscape	640 x 480, 72 DPI	
16" Portrait	640 x 870, 80 DPI	
21" Kong	1152 x 870, 72 DPI	

Convolution Address Translation

The convolution filtering to reduce flickering on interlaced displays must be performed transparently to the processor. This requires an address translation in the system controller to map the processors address space into the convolved address space. A derivation of the convolution address translation is provided below for 24-bit/pixel, full-screen PAL (768 x 576) with video mask and scanline cursor memory. The same translation can be used for NTSC with extra memory distributed throughout the frame buffer.

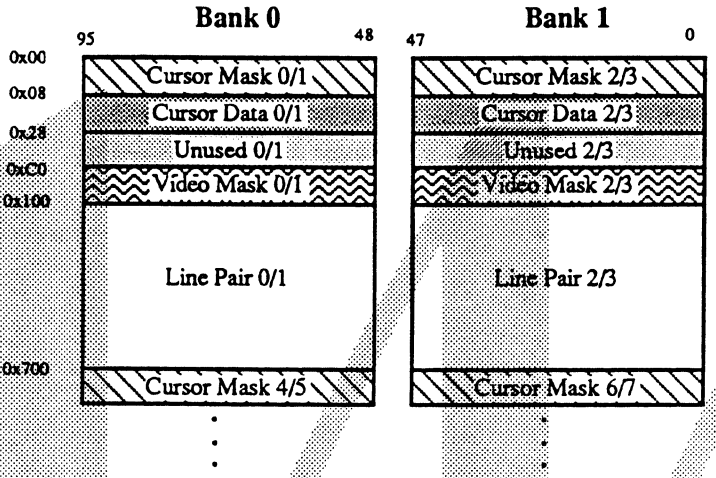
The software's view of the frame buffer in convolution mode is shown below. Each scanline in the frame buffer contains cursor information (both mask and image data), video mask information, and pixel information. The base address of the first scanline of pixel information is defined in the system controller by the Screen Address Register (SAR) and the amount of memory between scanlines, is screen row bytes (SRB). Both SAR and SRB must be 16-byte aligned.



Software's View of the Convolved Frame Buffer

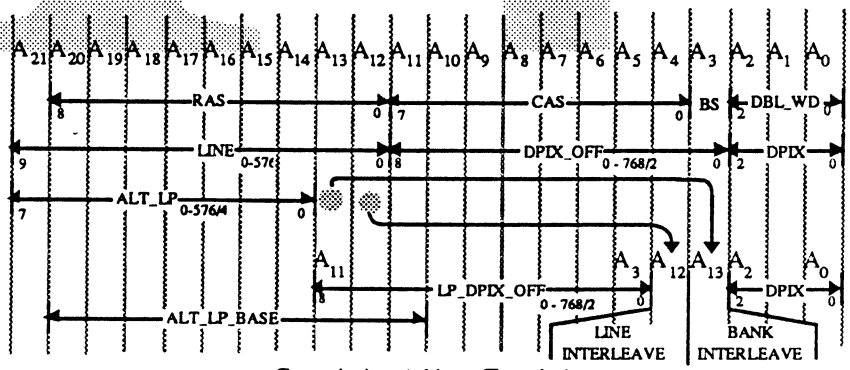
To allow the line number and pixel offset to be easily separated in hardware SRB is set to 1024 words in software: 128 words of cursor and video mask data; 768 words of pixel data; and 128 words of unused memory to make 1024 words. To recover the extra memory on each scanline distributed throughout the frame buffer (required for full-screen PAL), the address translation hardware must scale SRB to 896 words (a multiply by 7/8 -- a shift and an add). This has the effect of mapping the last 128 words of unused memory from the current scanline into the first 128 words of the next scanline. Overlapping the scanlines in this fashion doesn't create a problem because the software never accesses the last 128 words of each scanline address space, so the memory is not redundantly used by two scanlines.

The hardware's view of the frame buffer in convolution mode is shown below. Since the cursor and video mask information are mapped into pixel formats in the frame buffer and the address translation is performed to all read/write transaction to the frame buffer, all the information in the frame buffer is twisted into the format required for the filter. Treating everything as a pixel, successive scanlines are interleaved on a dual pixel basis to form line pairs and successive line pairs alternate between Bank 0 and Bank 1. This allows pixel information for four successive scanlines to clocked out of the video RAM serial ports in quick succession.

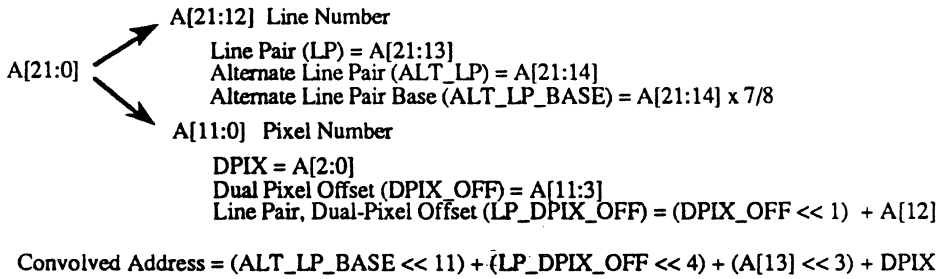


Hardware's View of the Convolved Frame Buffer

The linear address space, as viewed by software, can be broken down into a line number, a pixel number or offset in the scanline and a quad-word (pixel) select. To interleave two successive lines together into a line-pair, on a dual-pixel basis, the LSB of the line number is rotated below the dual-pixel offset. To alternate line pairs between double-word banks, the next line address bit is wrapped down into the A3. The remaining line address is then multiplied by 7/8 and added to the linepair, dual pixel offset as indicated in the diagrams below.



Convolution Address Translation



Convolved Address = (ALT_LP_BASE << 11) + (LP_DPIX_OFF << 4) + (A[13] << 3) + DPIX

Section 3

ELMER

Introduction

A brief overview of the functions performed by Elmer is provided below, followed by a more detailed description of the implementation of each function.

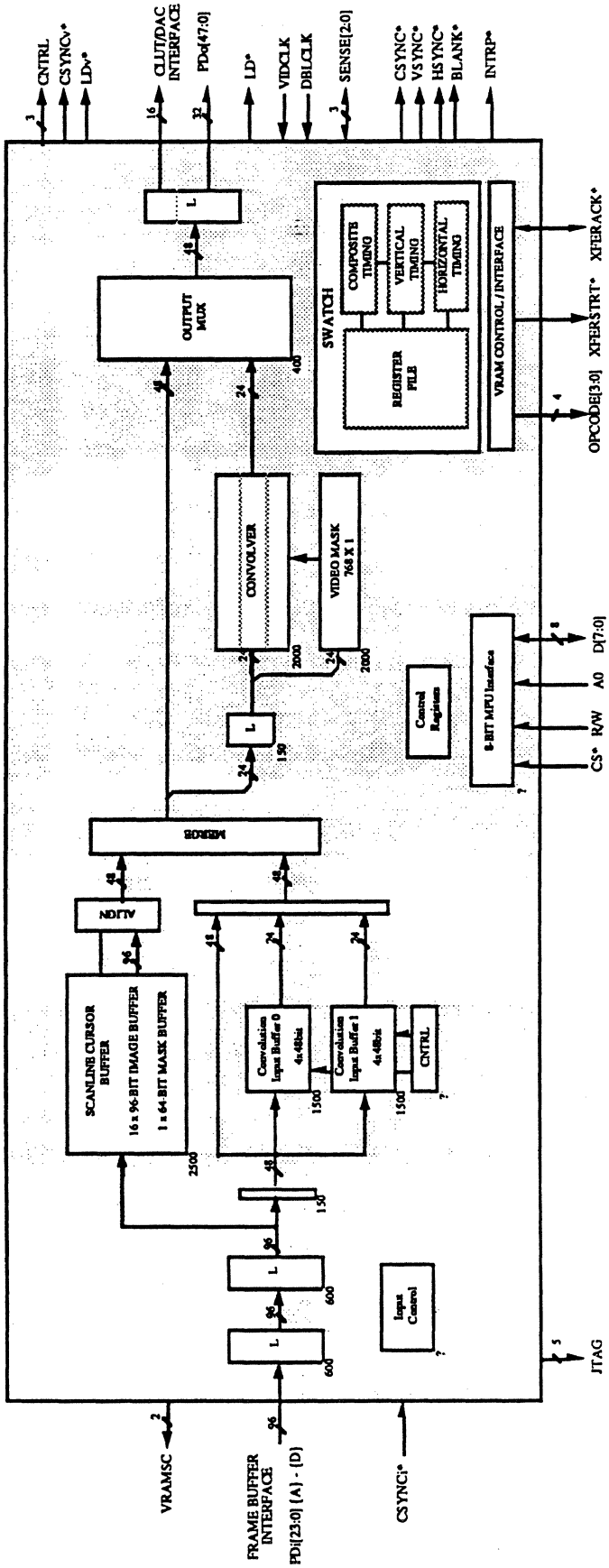
The primary purpose of Elmer is to serve as an interface between the Video RAM frame buffer an the CLUT/DAC and digital, NTSC/PAL video encoder (DENC) chips. The internal multiplexing, buffering and reorganizing of pixel data provides increased flexibility in frame buffer organization by decoupling the frame buffer pixel formats and transfer rates from that of the CLUT/DAC chip. This allows the frame buffer pixel format to be tailor specifically for JAG, rather than the CLUT/DAC chip.

The 96-bit frame buffer interface, four 24-bit pixel ports, is designed to interface directly to the Video RAM serial ports and transfer data at the maximum serial port clock rate, 33 MHz. The interface can be programmed to operate in 2:1 or 4:1 multiplexing mode. The 48-bit CLUT/DAC interface, two 24-bit pixel ports, is designed to support TTL transfer rates up to 57 MHz. The interface can be programmed to operate in several different modes and pixel formats. It is compatible with the 8-bit Chunky and 24-bit RGB pixel formats defined in the AC/DC spec and supports true-color and grayscale formats of a preliminary true-color CLUT/DAC chip, PRISM. The aggregate throughput of Elmer is designed to support 100 MHz, 24-bit/pixel, true-color and 220 MHz, 8-bit/pixel, grayscale video bandwidths. This is sufficient for a 21" true-color or 16", 144 DPI grayscale monitor.

Elmer contains a programmable video timing generator, SWATCH, which supports Apple's current and future monitor line. The VRAM interface block between Elmer and the system controller synchronizes the video timing generation with the video address generation. The interface allows Elmer to interrupt the system controller to initiate row transfers to the VRAM required for video refresh. Additionally, the Video RAM serial port control allows video mask and cursor information to be buffered into the device on a scanline basis during horizontal blanking.

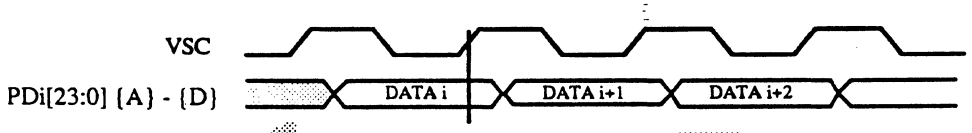
Elmer supports 24-bit/pixel, true-color, convolution filtering to reduce flicker in interlaced video. A dual set of input buffers provides an efficient true-color convolution implementation specific to JAG's frame buffer organization. A 768 x 1-bit video mask buffer is contained on chip to control the convolution filter on a pixel by pixel basis. A single scanline of the video mask data is loaded into the buffer during horizontal blanking. The video mask indicates regions of the scanline that contain video data and that should not be filtered. This allows the image quality of the video data to be maintained by passing the video data through unfiltered, while applying the convolution filter to computer generated graphics.

ELMER Block Diagram



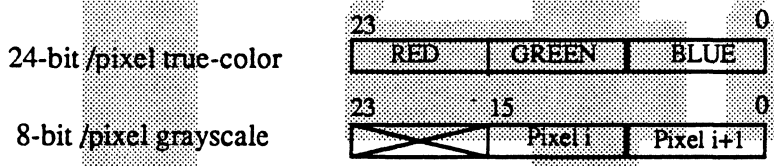
Frame Buffer Interface

The frame buffer interface is designed to directly interface to the serial ports of the Video RAM. The 96-bit input is logically divided into 4 24-bit pixel ports, PD[23:0]{A} - {D}. On the rising edge of the VRAM serial clock, VSC, pixel data is latched into the device.



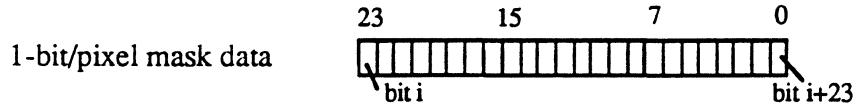
The interface can be programmed to operate in a 4:1 or 2:1 multiplexing mode through an internal control register. In 4:1 multiplexing mode, pixel ports{A} - {D} are latched into the device on the rising edge of VSC. The {A} pixel data is processed first, followed by the {B} pixel data, etc., until all the pixel data has been processed at which time the cycle repeats. This requires that the screen base address and screen row bytes be 16-byte aligned such that the first pixel of each line be aligned to the {A} port. In 2:1 multiplexing mode, pixel ports{A} and {B} are latched into the device on the rising edge of VSC, and pixel ports {C} & {D} are ignored.

The internal input multiplexing of the pixel ports supports two pixel formats, as shown below: an 8-bit/pixel grayscale format and a 24-bit/pixel true-color format. In the 8-bit/pixel format, two pixels are latched in each 24-bit pixel port with the most significant byte ignored. Any other type of data is mapped into one of these two formats.



An additional input format may be added to support a 32-bit bank of VRAM, in which four 8-bit pixels are selected sequentially from a pair of input ports (ie. 24 bits from {A} and 8 bits form {B}).

Video mask data clocked into the device during horizontal blanking is mapped into the 24-bit/pixel format as shown below. Bit 23 of the pixel port contains the least significant mask bit.



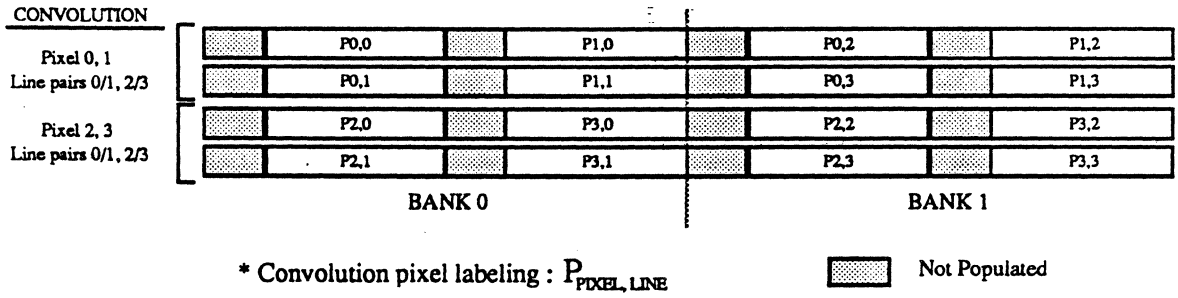
Convolution

To reduce flicker due to graphics data on an interlaced display, Elmer implements Apple's Convolution filtering on 24-bit true-color pixels. The FIR filter is applied to pixels from three successive scanlines according to the formula

$$(2 + \langle \text{line n-1} \rangle + 2 * \langle \text{line n} \rangle + \langle \text{line n+1} \rangle) / 4$$

To maintain the image quality of video data, the filter is only applied to graphics data. To control the convolution filtering, Elmer contains a video mask which identifies each pixel as video or graphics data. The mask data is loaded into 768 x 1-bit video mask buffer on a scanline basis during horizontal blanking and turns the filtering off whenever video data is being displayed.

To support convolution, the frame buffer is partitioned into two 64-bit wide banks. Successive scanlines are interleaved on a dual pixel basis to form line pairs. Successive line pairs then alternate between the even and odd banks, such that line pairs 0/1, 4/5, 8/9, ... are stored in BANK 0 and line pairs 2/3, 6/7, 10/11, ... are stored in Bank 1. This allows pixels from four adjacent scanlines to be clocked out of the video RAM serial ports in quick succession. The pixel data stream is illustrated in the diagram below.



Input Buffering and Reordering

In convolution mode, the device can only be operated in 4:1 multiplexing mode. Pixel data is clocked into the device at four times the standard interlaced bandwidth. The 4 24-bit pixel ports are logically combined to form 2 48-bit line pair ports. Pixel ports {A} and {B} are combined to form the EVEN line pair port and pixel ports {C} and {D} are combined to form the ODD line pair port. Each line pair port latches a dual-pixel chunk of the interleaved line pairs on the rising edge of VSC.

Elmer contains a pair of convolution input buffers to resequence the line pair data for the convolution filter. Each buffer contains 8 24-bit pixels, two pixel chunks from four adjacent scanlines. The buffers are operated in a ping-pong fashion, such that while one buffer is being loaded from VRAM the other is supplying data to the filter.

Internally, the pixels in the convolution buffers are reordered in an order convenient for the filter. The pixel above and below the pixel being displayed are sequenced to the filter in the following order: line <n-1>, line <n>, line <n+1>. The pixel data for the fourth line is ignored. It is not shown, but the pixel reordering changes in an alternating fashion on a line-to-line basis, as well as on a field basis.

Loading Video Mask

The video mask data to control the convolution filtering is loaded into the device during horizontal blanking. In the simplest case, only a single scanline of video mask data is required to turn the filtering on or off. A problem arises when the filtering is applied across a video/graphics boundary because the pixel above/below a video pixel may contain graphics data and create weird edge effects. The solution is to apply the filtering when anyone of the three pixels being convolved contains graphics data. This insures that a component of the graphics data will bleed into the video data and reduce potential flickering. To eliminate extra video mask buffering, the video mask data for all three scanlines used in the filtering are combined into a composite video mask as the data is read in during horizontal blanking.

Video mask data is mapped into 24-bit pixel data as indicated previously. To simplify the hardware, 768 bits of video mask, 32 24-bit pixels, are stored at the beginning of each row in the frame buffer for both NTSC and PAL. This allows the video mask data for the 4 adjacent scanlines to be accessed

in quick succession. The mask data is stored just prior to the video data of each scanline such that only one row transfer operation is required during horizontal blanking to clock in the mask data and initialize the serial ports for video refresh. After the video mask data is buffered into the device, the serial ports will be at the beginning of the line for active video. The transfer of all four scanlines of mask data requires 32 serial port accesses at the standard, convolution video refresh bandwidth; it is completed in 32 dotclks. The mask data is buffered in the convolution input buffers and resequenced similar to pixel except that the final destination is the video mask buffer.

CURSOR

The implementation of a cursor / overlay region is tentative at this point and should be considered as an optional feature. A software fallback position must be provided to handle the cases not supported by hardware (ie. multiple cursors, anti-aliased cursors, full-screen cursors). The implementation presented provides a generic mechanism for overlaying a variable size, rectangular region over active video refresh, but the goal is to provide at least a 16 x 16 Color Quickdraw cursor for all monitors. The size of the overlay region is primarily limited by the on-chip buffer for a single scanline of cursor information.

Overview

Given the fixed size and organization of the JAG frame buffer, extra memory is available in the frame buffer to store a rectangular, cursor region. For some monitors very little memory is available, for others, over 0.5MBy. The cursor region is stored in the extra frame buffer memory at the same pixel depth of the frame buffer, 24-bit/pixel true-color or 8-bit/pixel grayscale.

During horizontal blanking, a single scanline of the cursor region is clocked out of VRAM serial ports into a cursor scanline buffer on Elmer. The cursor scanline buffer consists of a 64 x 24-bit cursor data buffer and 64 x 1-bit cursor mask buffer which can support the following cursor sizes:

64 x 64 at 24-bit/pixel, true-color
16 x 16 at 24-bit/pixel, true-color, convolved
64 x 64 at 8-bit/pixel grayscale

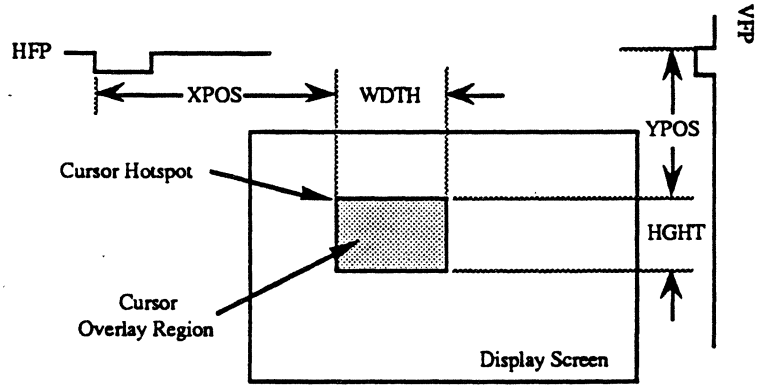
(Note: The cursor size in convolution mode is limited to 16 x 16 because three scanlines of cursor information are required to properly convolve the cursor. Also, the cursor size for some monitors may be limited due to available memory in the frame buffer.)

The cursor overlay region can be arbitrarily positioned on the screen by changing a few control registers (via a channel program - eek!). When the video refresh traces across the overlay area, the contents of the scanline buffer are merged with the standard video refresh (background image).

As defined for the current color cursor, the display of the cursor involves a relationship between the mask and the image. The data pixels within the mask (bit==1) replace the background pixels. The data pixels outside the mask (bit==0) are displayed using an XOR with the background pixels. If data pixels outside of the mask are white (R=G=B=0xFFFF) the pixels are transparent an the background image is unchanged. If pixels outside the mask are black (R=G=B=0x0000), the two most significant bits of each component of the background pixel are complemented. This guarantees at least a 25% contrast between the background and complemented cursor region. All other values outside of the mask cause unpredictable results.

Cursor Region Control

The rectangular, cursor region is defined in memory by a base address and rowbytes. The base address points to the beginning of the first line *displayed* and rowbytes is the offset between scanlines. Both the base address and rowbytes must be 16-byte aligned. The size of the cursor region on the screen is defined by two registers, height (HGHT) and width (WDTH). The height and width only change when the region is completely redefined. The position of the cursor hotspot, the upper-left hand corner, is controlled by the x position (XPOS) and y position (YPOS) registers. The x and y position of the cursor hotspot is defined relative to the beginning of the horizontal and vertical front porch, respectively. Both XPOS and YPOS are 12bit, signed values which allows the region to positioned off screen.



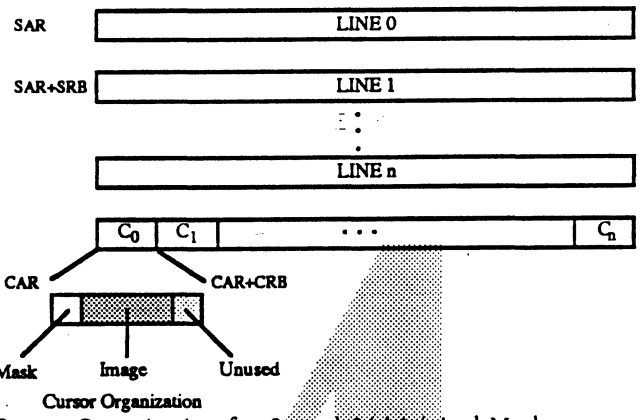
To simplify the cursor address generation hardware required for the VRAM serial port control, if the cursor region moves off the top of the screen, then the cursor base address register, CAR, in the system controller must be updated such that it points to the beginning of the first line displayed.

Cursor Refresh

The cursor region refresh process is similar to the standard video refresh except that cursor information is only fetched when the cursor region covers a portion of the current scanline. During horizontal blanking, the cursor information for the next scanline is clocked out of the VRAM serial ports into a scanline buffer. The address generator in the system controller provides the same functionality for both the cursor and video address generation. When the first line of the cursor region is active, the VRAM interface provides an opcode to the address generator to load the transfer cursor address register, TCAR, with the cursor base address, CAR. The system controller performs a VRAM read-transfer memory cycle to TCAR and split-read transfer cycles as required to maintain the shift register. The serial ports are clocked until a single scanline of cursor, as defined by the width register, is buffered into the device. Guidelines will be developed such that software can insure that there is enough time during horizontal blanking to reload the VRAM shift registers for active video and buffer the convolution video mask when appropriate. On successive scanlines, TCAR is incremented by the cursor rowbytes, CRB, to refresh the cursor region on a line-to-line basis.

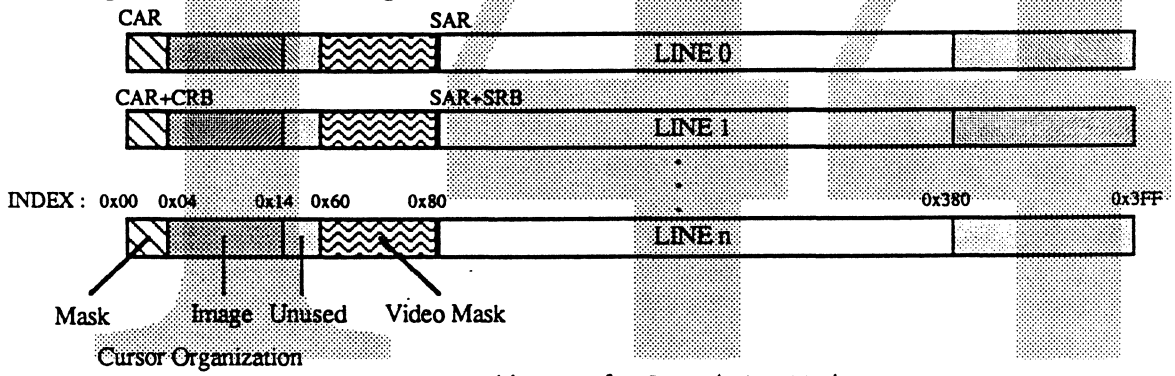
The cursor image data is stored in the extra frame buffer memory at the same pixel depth of the frame buffer. The 1-bit/pixel cursor mask data uses the 1-bit/pixel format. The cursor mask and image data for each scanline must be organized sequentially in memory. For each scanline, 96-bits of mask data is clocked into the buffer, followed by up to 64 pixels of image data, as defined by the

cursor width register. In 24-bit/pixel true-color and 8-bit/pixel grayscale modes, successive scanlines of cursor information are grouped sequentially in memory, as shown below. Each scanline of cursor data must be 16-byte aligned, as defined by CAR and CRB.



Cursor Organization for 8- and 24-bit/pixel Modes

Due to the address translation in convolution mode, the scanline cursor information is grouped with the video mask and graphics data for each scanline, distributing it across successive scanlines in the frame buffer. In this case, the cursor rowbytes, CRB, would equal the screen rowbytes, SRB. The cursor organization and addressing in convolution mode is shown below.



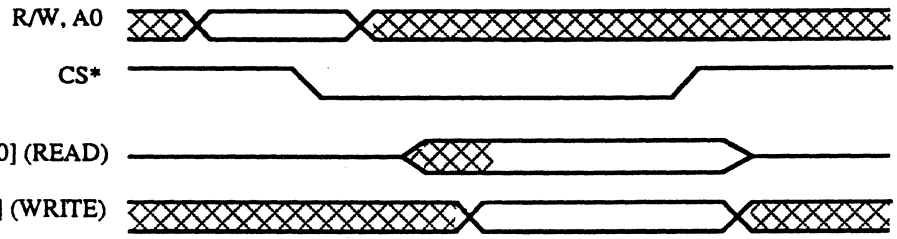
Cursor Addressing for Convolution Mode

MPU Interface

Elmer provides a standard 8-bit MPU interface to access all internal control and status registers. All registers are indirectly addressed by first loading an internal, 8-bit index register. Address bit A0 is decoded to select the internal index register (A0=0b) or the register space (A0=1b). To support block read/write by Mazda, the index register is automatically incremented after every access to the register space. When the index register reaches 0xFFH, it will roll over to 0x00H on the next access.

A0	R/$\overline{W}$	Bus Operation
0	0	Load Internal Index Register (Index)
1	1	Read Register (Index) Increment Index Register (Index++)
1	0	Write Register (Index) Increment Index Register (Index++)

The timing for the MPU interface is shown below.



Video Timing Generation

To allow the motherboard to support a wide range of monitors, both MAC and JAG monitors, in a consistent fashion, a programmable video timing generator developed in SEG will be ported to Elmer to generate the sync and blank monitor timing. A detailed discussion of the video timing generation is not presented in this section, rather only a brief description of the interface between the video timing and address generation is presented. For a detailed description of video timing generation, refer to Appendix A, "Stopwatch Theory of Operations, Version 3.1".

The video timing generator has 4 major components: horizontal timing circuit, a vertical timing circuit, a composite timing circuit, and a large register file to store timing parameters for all three timing circuits. The horizontal timing circuit can be in one of four states: the front porch, the sync pulse, the back porch, and active video. A pixel counter maintains a count of how many pixels have passed in the current horizontal state and is compared to the horizontal timing parameters to transition between horizontal states (front porch, sync, ... etc). Similarly, the vertical timing circuit can be in one of four states : the front porch, the sync pulse, the back porch and active video. A half line counter maintains a count of how many pixels have passed in the current vertical field state and is compared to the vertical timing parameters to transition between vertical states. The composite timing circuit is concerned with generating the proper equalization pulses and serrations, and compositing them with vertical and horizontal sync/blank pulses to generate CSYNC~/CBLANK~ output. SWATCH contains additional circuitry to lock on to the horizontal sync of an external signal and synchronize the horizontal and vertical timing circuits to compensate for pipeline delays and to maintain correct status information

The current SWATCH video timing generator is flexible enough to handle existing and future monitors. HDTV will not be supported because the CLUT/DACs do not support a tri-level sync. An area of concern is the frequency of operation for a high DPI grayscale monitor. For higher frequency monitors, the horizontal timing circuit is clocked at a divided down dotclk and thus maintains a count of how many multiple pixel chunks have passed in the current horizontal state. For a 220 MHz Hi-DPI monitor, if the circuit can not run at 55 MHz, the transitions between horizontal states will be constrained to 8 pixel boundaries, but this should not be a problem.

Vertical Line Interrupts

To help Wilson avoid frame tears when transferring regions of data, Elmer provides eight vertical line interrupts that can be programmed to occur at any vertical raster position. The 12-bit vertical line interrupt registers define a line count relative to vertical front porch. When any one of the 8 vertical line interrupt registers equals the current line count, as maintained in the vertical timing generator, a vertical line interrupt signal, LNINTRP, is asserted to Mazda for n clock cycles. Mazda latches the line interrupt signal and passes it on to the XJS.

To allow XJS to quickly service the line interrupts, Mazda maintains an internal line count of the vertical beam position relative to the vertical sync pulse. Thus, XJS can quickly read the line count from Mazda and identify the interrupt source. The line count is controlled by Elmer via the CLEAR and COUNT signals. CLEAR is connected to VSYNC~ to clear the counter on the falling edge of the vertical sync pulse. COUNT is connected to HSYNC~ to increment the counter on the falling edge of the horizontal sync pulse.

Each of the 8 vertical line interrupts can be independently masked in Elmer through an 8-bit interrupt mask register. To update the mask or line count register in Elmer, XJS must initiate a channel program to transfer the data through Mazda. The theoretical worst case latency for this transfer to occur is one vertical line, or 14uS for the Kong or Portrait monitor. In actuality, the vertical line interrupts would never be grouped this close together, allowing for more margin. Overhead and latency to update interrupt control registers on Elmer is an issue of concern.

Video RAM Address Generation and Serial Port Control

The address generation for video and cursor refresh is partitioned across two chips: the system controller and Elmer. The system controller contains the registers and data path for the screen and cursor address generation. Elmer contains the control logic for the address generation data path. Elmer is responsible for initiating all read-transfers to the VRAM shift registers by providing a 4-bit opcode to the system controller which is decoded to control the address generation and the VRAM read-transfer memory cycle. After a row transfer has been initiated, it is the system controllers responsibility to initiate split-read transfers, as required without Elmer's knowledge, to maintain the serial port data. Refer to the Memory ERS for details on the register and opcode definitions.

The 4-bit opcode encodes 6 types of operations for cursor and video refresh as defined below:

1) INIT - XFER	Initialize transfer address registers (TAR) and perform read-transfer (RT)
2) INC RB/2 - XFER	Add ROWBYTES/2 to TAR and perform read-transfer
3) INC RB - XFER	Add ROWBYTES to TAR and perform read-transfer
4) XFER B0 - XFER B1	Perform read-transfer to banks independently
5) XFER B0 - INC RB - XFER B1	Read-transfer Bank 0, add ROWBYTES to TAR, Read-transfer Bank 1
6) XFER B1 - INC RB - XFER B0	Read-transfer Bank 1, add ROWBYTES to TAR, Read-transfer Bank 0

The first two operations are performed during Vertical blanking to initialize the address generators to the base address of the frame buffer or cursor memory. For interlaced video without convolution filtering, ROWBYTES/2 is added to the base address for the ODD field. During horizontal blanking, the third operation updates the address registers to the beginning of the next line. Operations 4 through 6 are used for interlaced video with convolution. In convolution mode, the frame buffer is

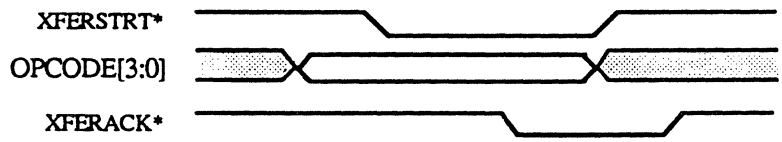
seen as two 64-bit banks which must be independently controlled. The three operations perform transfers to each bank separately and updates the transfer address register to the beginning of the next line-pair in an alternating fashion between banks. A summary of how the operations would be used to control the cursor and video address generation is shown below.

Non-Interlaced Displays:
During VBLANK: INIT - XFER
HBLANK: INC RB - XFER

Interlaced Displays w/o convolution:
During VBLANK (EVEN Field): INIT - XFER
(ODD Field): INIT - XFER
INC RB/2 - XFER
HBLANK : INC RB - XFER

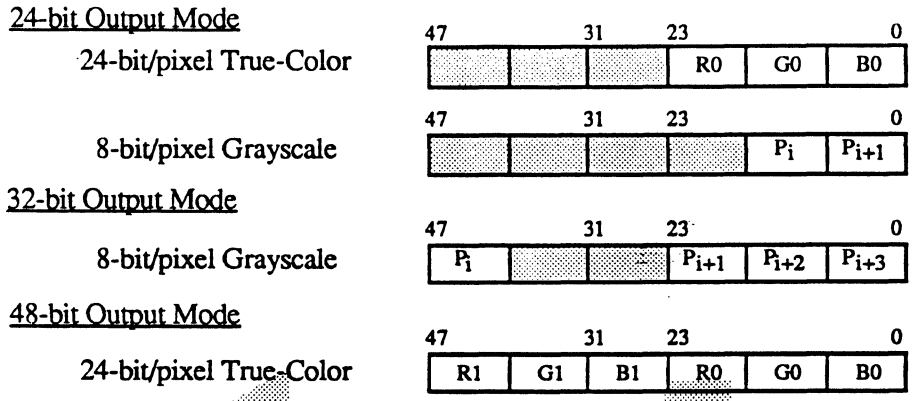
Interlaced Displays w/ convolution:
During VBLANK: INIT - XFER
HBLANK: XFER B0 - XFER B1
HBLANK: XFER B0 - INC RB - XFER B1
* Depending on the line-pair grouping, the operations alternate for successive HBLANKs

The interlocked handshake interface between Elmer and the system controller is illustrated below. The read-transfer cycle is initiated by supplying an opcode and asserting the XFERSTRT* signal. The system controller acknowledges the completion of the read-transfer cycle by asserting XFERACK* until the XFERSTRT* signal is deasserted. The XFERACK* signal is bi-directional and provides slow serial clock, divided by four, to the system controller when XFERSTRT* is deasserted.



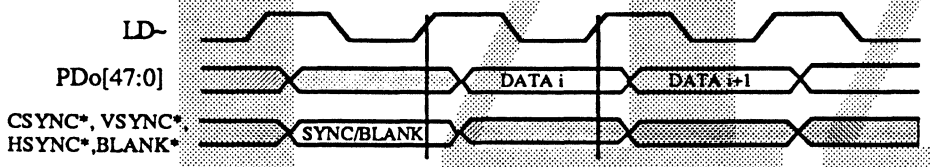
CLUT/DAC Interface

The CLUT/DAC interface is designed to support TTL transfer rates of up to 57 MHz. The interface can be programmed for grayscale or true-color pixel formats with 24-bits, 32-bits or 48-bits valid. This supports true-color video bandwidth up to 110 MHz and grayscale video bandwidth up to 220 MHz. The 32-bit grayscale format is provided for compatibility with the 8-bit Chunky pixel format defined by AC/DC. (Note: The awkward byte lane positioning is to provide footprint compatibility between 32-bit and 48-bit CLUT/DAC interfaces). The pixel formats and output modes are illustrated below.



Output Pixel Formats

As illustrated in the figure below, the output pixel data, PDo[47:0], SYNC and BLANK data are valid on the rising edge of LD~. LD~ is generated from either VIDCLK, divided down from the differential ECL clock, or from the TTL clock input, DBLCLK, and is used to latch the pixel data and SYNC and BLANK information into the CLUT/DAC chip. Refer to the clock generation section for an explanation of the VIDCLK and DBLCLK signals.



For composite output, CSYNC information is latched into the CLUT/DAC chip and is internally pipelined to maintain timing with respect to the pixel information. For dot clocks over 50 MHz, separate CSYNC, VSYNC and HSYNC signals bypass the CLUT/DAC chip to reduce jitter in the SYNC timing. Thus, for the Kong, Portrait, 16" Landscape and Hi-DPI portrait monitors, the SYNC information must be internally delayed in ELMER to compensate for the pipeline delay in the CLUT/DAC chip. This can be performed by taking the pipeline delay into consideration when programming SWATCH.

Monitor Sense Lines

The monitor sense lines are used to uniquely identify the type of monitor connected to the frame buffer. Sense[2:0] are bidirectional signals with open drain outputs and are mapped into the RDSENSE and WRSENSE control registers. By driving a single output low and reading the two other resultant signals, and repeating the process for all three signals, software can determine the monitor connected to the frame buffer and generate the proper video timing. The sense lines must be continuously monitored to insure the low frequency SYNC timing for NTSC/PAL is not connected to a non-interlaced monitor.

After system reset, the sense lines come up in input mode. The current state of the sense lines can be determined by reading the RDSENSE status register. The sense line inputs are sampled after each write to the WRSENSE register. After initialization, hardware will determine if the sense lines change unexpectedly (the user disconnecting the monitor cable) and blank the CLUT/DAC output and gate

the syncs until the sense lines return to the initialized state or are re-initialized. This is intended to protect the monitors form improper timing and to minimize EMI.

Control / Status Registers

A rough cut at the control and status registers for Elmer is provided below. The video timing registers are as defined in the "Stopwatch Theory of Operations, v3.1" .

A channel program is required to read/write any registers through Mazda. The MPU interface and register addressing will be defined in such a fashion so as to allow block transfers of logical groups of registers, ie. timing registers, interrupt registers, control and status registers.

Stopwatch registers	
Horizontal timing control --	
Horizontal Front Porch	12 bits
Horizontal Sync Pulse	12bits
Horizontal Back Porch	12bits
Horizontal Active Pixels	12bits
Pixels to half line	12bits
Horizontal Pixel start number	12bits
Horizontal Counter Load	1bit
Horizontal Counter Stop	1bit
Horizontal Timing Halt	1bit
External HSYNC Enable	1bit
HSYNC~ Active High	1bit
Vertical timing control --	
Vertical Front Porch	12 bits
Vertical Sync Pulse	12bits
Vertical Back Porch	12bits
Vertical Active Lines	12bits
Vertical Line start number	12bits
Vertical Counter Load	1bit
Vertical Timing Halt	1bit
VSYNC~ Active High	1bit
Composite timing control --	
Pixels to Serration	12bits
Equalizing Pulse Enable	1bit
Serration Enable	1bit
Reset control	
Soft Reset	1bit
Status Registers --	
Odd Field State	1bit
Horizontal State:	
Front Porch	1bit
Sync Pulse	1bit
Back Porch	1bit
Active Pixel	1bit
Vertical State:	
Front Porch	1bit
Sync Pulse	1bit
Back Porch	1bit
Active Line	1bit
Horizontal Line Count	12bits
Vertical Line Interrupts	
Line Interrupt 1	12bits
Line Interrupt 2	12bits
Line Interrupt 3	12bits

Line Interrupt 4	12bits
Line Interrupt 5	12bits
Line Interrupt 6	12bits
Line Interrupt 7	12bits
Line Interrupt 8	12bits
Interrupt Mask	8bits
Convolution	
Convolution Enable	1bit
Video Mask Disable	1bit
Convolution Buffering Disable	1bit
VRAM interface	
2:1 MUX mode Enable	1bit
Grayscale Pixel format Enable	1bit
CLUT/DAC Interface	
Grayscale Pixel format Enable	1bit
32-bit Output Mode Enable	1bit
48-bit Output Mode Enable	1bit
Cursor	
x position, XPOS	12bit (signed)
y position, YPOS	12bit (signed)
cursor height, HGHT	12bit
cursor width, WDTH	12bit
Transparency Disable	1bit
Sense Lines	
Sense Line Read Access (RDSENSE)	3bits
Sense Line Write Access (WRSENSE)	3bits

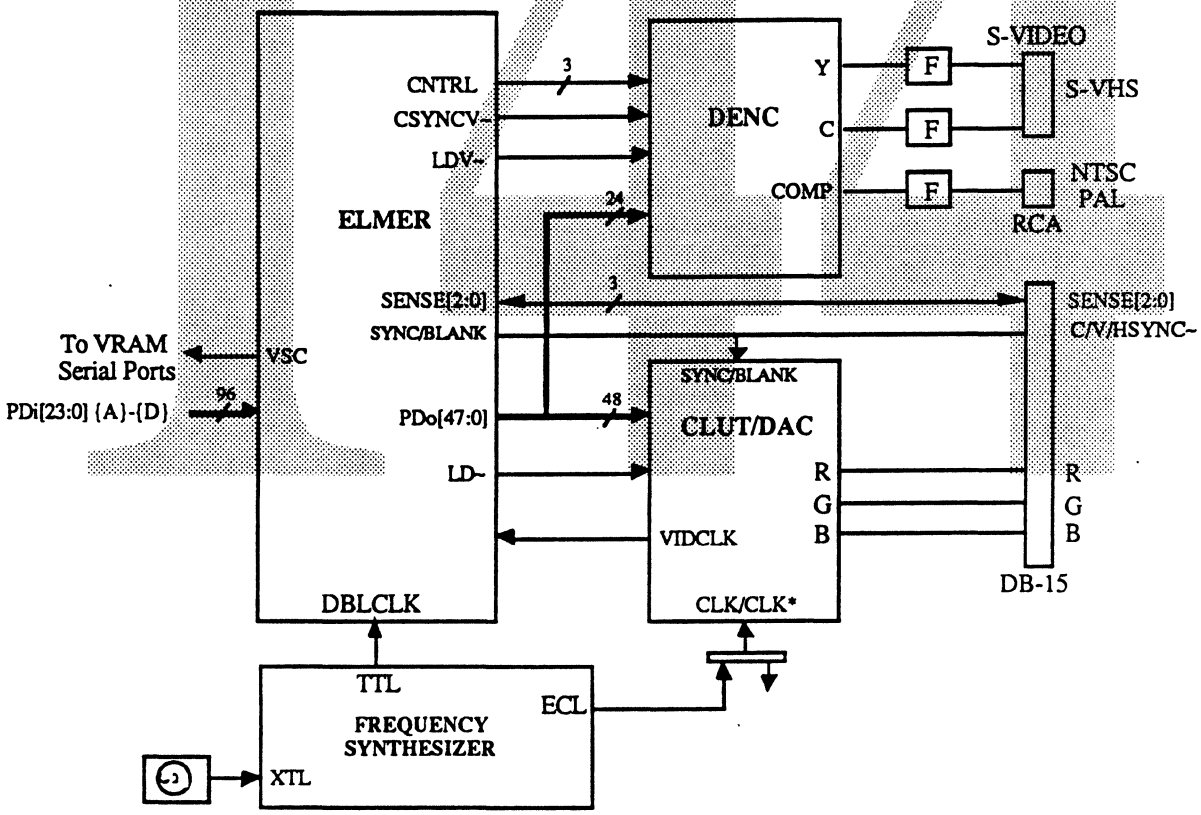
Pinout

VRAM Interface and Control			
96	I	PDi[23:0] {A}-{D}	VRAM Interface
2	O	VRAMSC	VRAM Serial Clocks
1	O	XFERSTRT*	Serial Port Transfer Start
1	I/O	XFERACK*	Transfer Ack. / VSC
4	O	OPCODE[3:0]	Transfer Operation
Video Output Interface			
48	O	PDo[47:0]	CLUT/DAC Interface
1	I	VIDCLK	Divided Down Pixel Clock
1	I	DBLCLK	TTL Pixel Clock
1	O	LD~	CLUT/DAC Load Clock
1	O	LDv~	DENC Load Clock
Video Output Timing and Control			
3	I/O	CNTRL	DENC Serial Port Control
1	O	CSYNCv~	DENC Composite Sync
1	O	CSYNC~	CLUT/DAC Composite Sync
1	O	VSYNC~	CLUT/DAC Vertical Sync
1	O	HSYNC~	CLUT/DAC Horizontal Sync
1	I	CSYNCi*	External Composite Sync
3	I/O	SENSE[2:0]	Monitor Sense Lines
1	O	LNINTRP*	Vertical Line Interrupt
MPU Interface			
8	I/O	D[7:0]	MPU Data Bus
1	I	A0	Address Select
1	I	R/W~	Read/Write
1	I	CS~	Chip Select
Misc			
1	I	RESET	System Reset
5	TBD	JTAG	Boundary Test
26	I	Power/GND	
211 Total			

Section 4

Video Output

To meet the user demands of a multi-media machine, JAGUAR provides two sources of video output on the motherboard: a high-resolution, non-interlaced component RGB output; and an encoded, NTSC/PAL video output. The goal of providing both graphics and video outputs on the motherboard is to provide a highly integrated, low cost solution to allow the user to record or project the display using standard video equipment. As illustrated in the diagram below, the CLUT/DAC chip generates the component RGB output for the high-resolution, non-interlaced monitor and a digital NTSC/PAL encoder (DENC) chip generates component Y/C and composite NTSC/PAL video output for standard video equipment.



The CLUT/DAC and DENC share a common pixel data input and refresh hardware such that only one output is active at a time. To prevent garbage form being generated from the inactive device when the CLUT/DAC is used to drive a graphics monitor, the DENC clock is gated to disable the output. Similarly, when the DENC is used to generate the NTSC/PAL video output, BLANK* is asserted to blank the graphics monitor and the component RGB SYNC signals are gated.

The transition between the graphics and video outputs is under software control. The video timing generation and control registers in ELMER, and the video address generation parameters in the system controller, must be updated to support the different monitor timings. Also, Quickdraw's view of the frame buffer must be updated to support the different frame buffer format. Current system software does not provide this functionality under multifinder without rebooting. Local copies of global parameters stored off of (A5) (ie. base address, rowbytes, ...etc.) can not be dynamically updated and the machine must be rebooted.

NTSC/PAL Video Output

The digital encoder (DENC) chip provides a low cost, consumer quality, encoded video output capability. It generates component Y/C (S-VHS) and composite video encoded to NTSC and PAL standards. No GENLOCK support is provided. DENC supports a 24-bit digital interface compatible with the CLUT/DAC, requires no adjustments in manufacturing and uses minimal external components.

DENC performs the entire encoding process in the digital domain before performing the digital to analog conversion. The 24-bit RGB data is converted to YUV data and digitally modulated into the NTSC/PAL format. Triple on-chip DACs (of sufficient resolution) generate the final Y/C and composite analog outputs.

Third-order elliptical filters are required on the outputs for sample-hold elimination and to attenuate glitch impulses and aliased frequency components. The filters can be constructed with passive, surface mount components. Finally, the outputs are AC coupled into 75 ohm termination to eliminate any signal when the NTSC/PAL output is not being displayed and DENC is inactive.

The 24-bit pixel data, PDo[23:0], and composite sync, CSYNCV~, is latched into the device on the rising edge of LDV~, similar to the CLUT/DAC interface. Separate load and sync signals are required to isolate the two output blocks. Separate load clocks are required to protect DENC during non-interlaced modes of operation and separate CSYNCs are required to prevent the non-interlaced monitor from free running when the encoded output is being used. Special attention to the layout will be required to minimize loading on the data port. Also, setup and hold requirements will be defined to be compatible with AC/DC to maintain the high transfer rates required for the CLUT/DAC.

Component RGB Output

The component RGB output is generated by the AC842 CLUT/DAC chip, commonly referred to as AC/DC. AC/DC is an Apple proprietary, triple 8-bit CLUT/DAC chip which supports up to 100 MHz grayscale video and 64 MHz true-color video bandwidths. This is sufficient for a 21", 1152x870 grayscale KONG monitor and a 16", 830x630 color landscape monitor (preliminary). AC/DC has a 32-bit pixel bus interface capable of sustaining TTL transfer rates of up to 64 MHz. AC/DC supports numerous pixel formats of which only the 8-bit Chunky and 24-bit/pixel formats are used.

Because gamma correction can not be performed in the CLUT prior to the convolution filter, the convolution circuitry in AC/DC is not used. Instead, 24-bit/pixel true-color convolution filtering is performed in Elmer and the triple 256x8 CLUT is used for gamma correction only.

The output video is compatible with RS343A voltage levels with a programmable blank pedestal of 0 or 7.5 IRE units. For grayscale over 50 MHz operation, SYNC information bypasses the CLUT/DAC and the output video is generated by the BLUE channel.

A Built in Logic Block Observation (BILBO) circuit in the CLUT/DAC allows software to perform signature analysis of the video data immediately prior to the DACs.

The 8-bit MPU interface is connected to Mazda to access the CLUT and internal control and status registers. A channel program must be generated to read/write the CLUT and registers in block mode fashion. To prevent frame tears, the CLUT should only be updated during vertical blanking.

For complete information on the AC/DC CLUT/DAC chip, refer to the AC842 Data Sheet.

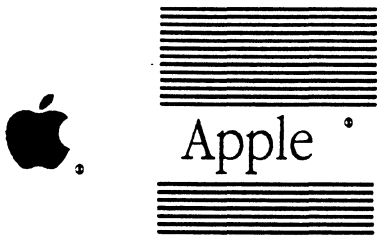
A possible alternative to AC/DC is PRISM, a preliminary CLUT/DAC chip under development. PRISM is a non-proprietary CLUT/DAC chip developed specifically for JAGUAR. PRISM contains a triple 256 x 10 CLUT and triple 10-bit linear DACs for full gamma correction with no reduction in color space. The 48-bit pixel bus interface supports TTL transfer rates up to 64 MHz to maintain 100 MHz true-color and 165 MHz grayscale video bandwidths. (Note: The monitors supported on the motherboard are constrained by the fixed frame buffer size and are limited to 100 MHz grayscale and 64 MHz true-color.)

Clock Generation

To support a wide range of monitors, the desired dot clock frequency is generated by frequency multiplication of an inexpensive TTL oscillator via a PLL frequency synthesizer. Both the Apple/AD 901, Endeavor, and the National Semiconductor DP8531 frequency synthesizers produce a programmable frequency (M*CLK/N) differential ECL clock output, CLK/CLK*, and a TTL clock output, DBLCLK, which is divided down by 2 from CLK/CLK*.

As illustrated in the diagram previously, the two clock sources are used in mutually exclusive modes of operation to clock all of the video timing. When pixel data is latched into the CLUT/DAC chip at less than 30 MHz, the internal dot clock of the CLUT/DAC chip is generated from the differential ECL clock inputs, CLK/CLK*. The CLUT/DAC chip provides a TTL clock output (max. 30 MHz), VIDCLK, divided down a programmable amount (1, 2, 4 or 8) from CLK/CLK*. When pixel data is latched into the CLUT/DAC chip at greater than 30 MHz, the differential ECL clocks inputs should be grounded and the internal dot clock is generated directly from the LD~ input. In this case, VIDCLK is invalid and the TTL clock, DBLCLK, is used to generate the video timing.

Currently, the TTL clock output of the Endeavor chip has a maximum clock frequency of 50 MHz, one half of the maximum ECL clock output of 100 MHz. This is insufficient for 24-bit/pixel PAL which requires a 59 MHz TTL clock for the convolution filter. An ECL to TTL converter would be required to use the Endeavor part. The National part can generate several TTL clock outputs of which the PCLK is rated at up to 64 MHz. Also, it contains two reference inputs to lock on to the HSYNC component of an external reference. This implementation requires further study because the 15 kHz horizontal loop frequency is slightly below spec and generates some horizontal jitter on the output.



Jaguar Decompression

External Reference Specification
Special Projects

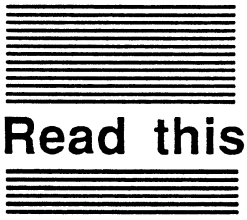
Draft - November 15, 1989

Return Comments to:
Juan Pineda
Phone: x4-9265
AppleLink: JUAN
MS: 60AA

144

Apple CONFIDENTIAL

Contents



Scope

This document presents a hardware acceleration architecture for video decompression on Jaguar. The architecture has been designed to work on the CCITT P*64 standard but is also designed to be programmable enough to handle other compression formats such as MPEG or Apple customized formats.

The architecture presented in this document is still preliminary, and there is much work ahead, especially in algorithm simulation. In addition, we are looking at alternatives of working with other companies to develop a decompression chip set for Jaguar.

Definitions

CCITT P*64	Video compression standard developed by CCITT standards committee targeted at video teleconferencing. Data rate is variable in 64K bit steps from 64Kbits to 1.92Mbits/second.
MPEG	Video compression standard under development targeted at high quality storage of moving images. MPEG is intended to be a superset of P*64.
DCT	Discrete Cosine Transform is a mathematical transform that converts an image to frequency domain. CCITT and MPEG compression algorithms are based on this transformation.
RGB, YUV	RGB and YUV are two commonly used representations for color. RGB represents a color by three scalar values that represent the intensity of red, green and blue components of a color. YUV represents a color by three scalar values also. Y represents the over all intensity, while U and V represent color difference between Y and two of the color components. RGB and YUV are related by a linear transform.

The problem

One goal of Jaguar is to provide full motion video as a fundamental data type. We envision that motion video will be used in recorded as well as interactive applications. Recorded applications would include the annotation of documents with video sequences or the perusal of a news event in a video data base. Interactive applications would include teleconferencing or the editing of a personal video tape sequence.

Unfortunately, full motion video has large bandwidth and storage requirements. For example, NTSC video requires about 50Mbytes/second of bandwidth, which would consume 180Gbytes of storage for one hour. Today's networks, and mass storage alternatives for personal computing provide about one megabyte/sec of bandwidth, and less than one gigabyte of storage.

Fortunately, there is much redundancy in video information, and it is possible to compress a video sequence in both space an time dimensions. The CCITT "P*64" standard is designed for video teleconferencing, and allows for the compression of a 30Hz 360x288 pixel image at a rate variable from 64Kbits/second to 2Mbits/second. The variable rate allows for the reduction of bandwidth by reduction of reproduction quality.

P*64 applied to NTSC video can yield an arguably reasonable quality reproduction at a bit rate of 1.25Mbit/second. This is a compression of 300 times the full bandwidth, and would lower the bandwidth requirement enough to allow compressed video to be transmitted on local area networks and stored on magnetic or optical mass storage. An hour of video would require 550Mbytes of storage, which is still not small, but is small enough to be stored on a single optical disk, or on hard disk.

There are other compression standards under development as well. JPEG is a still frame compression standard that has been developed over the past few years, and is reasonably stable. MPEG is a high quality motion video standard that is under development, and is still quite early in its evolution. MPEG is not currently well enough defined to begin complete hardware implementation. In addition to MPEG and JPEG, it seems likely, that Apple would want to develop it's own format for compression that would better meet the needs of personal computing applications.

It would be reasonable to expect sources of video information to come from many different sources, and to be encoded in all of these formats. With ISDN connection integral to Jaguar, P*64 encoded images would be transmitted through the phone networks. Higher quality recorded images encoded with MPEG or and Apple format would be stored on file servers, be transmitted over networks, or be available on optical disks.

It would seem that there is a terrible complication in this multiplicity of formats. Fortunately, all these compression formats are based on a DCT (discrete cosine transform) frame differencing techniques, and consequently are similar in the bulk of the computation that they perform. It seems possible that a single general purpose computation accelerator used in conjunction with software could be used for all these algorithms.

Unfortunately, because of the undefined state of MPEG, there is a risk that the hardware presented in this document may not be able to decode the final standard. It will be a race between our hardware schedule, and the evolution of MPEG. Our strategy here is to use expert advice to understand the range of possibilities and make the Jaguar decompression hardware programmable enough to handle the range.

While it would be highly desirable for all Jaguars to provide the ability to both encode and decode video in real time, it would be acceptable to make real time encoding an optional feature. Many of the applications, such as document annotation, data base perusal or editing can be performed with only a decoder. In addition, real time encoding would only be generally useful when used in conjunction with a video input digitizer for capturing the output of a camera, or other live video source. Encoding would most likely add an additional cost beyond the cost of the decoder because it requires about four times more computation than decoding. This additional cost added to the cost of a video digitizer would be hard to justify in all Jaguars.

In addition to the hardware architecture presented, we are looking at the alternative of working with other companies to develop a decompression chip set for Jaguar. This chip set would most likely have many of the attributes of the architecture presented here. In particular, another chip set would interface to Wilson on the same bus, and would share system memory for its memory requirements.

Goals

The Jaguar decompression accelerator will perform real time decompression of motion video.

The Jaguar decompression accelerator must be capable of decompressing CCITT P*64 encoded video. P*64 is important because it will be the standard for ISDN teleconferencing, and because it provides a usable encoding for NTSC video. In order to insure compatibility, the decompression accelerator must be able to operate at the maximum data rate for P*64 (P=30) of 2.4Mbits/second.

The Jaguar decompression accelerator must be programmable so that it can be used for encodings other than P*64. P*64 has limitations that make it less desirable for storage of video. In particular, it has no provisions for reverse playback and fast forward. For this reason, it would be desirable for the hardware to be programmable in order to support formats that have these features. At the very least, the accelerator should be able to be programmed to handle an Apple custom format that would provide such features.

It is a goal that the Jaguar decompression accelerator be able to operate on MPEG encoded images. We believe that MPEG will be an important standard for storage of motion video and will provide features and quality better than P*64 for stored video. Because of the early state in the evolution of this standard, this goal may be quite difficult to achieve.

It is a goal that the decompression accelerator cost less than $50 (manufacturing cost). Since decompression will be part of every Jaguar, it is essential that the decompression accelerator burden the cost of the base system as little as possible.

It is a goal that the decompression accelerator rely on software as much as possible. This will allow for programmability and minimize the cost of hardware. It is important that the decompression algorithms not use all of the CPU bandwidth so that other operations may be performed simultaneously with display of compressed video. For this reason, it is a goal that the decompression algorithm not use more than 50% of the bandwidth of a single CPU.

Architecture

The architecture for the decompression engine applies hardware to the computation intensive portions of problem, but leaves the more complex task of data formatting to software. So for example, a hardware engine is provided for the Huffman decoding of coefficient block data, but header extraction is performed in software.

Where possible, commodity VLSI is used for generic operations, while ASICs are used for system interface,and more obscure and unavailable functions. The most compute intensive part of the problem is the inverse DCT, and it is performed by a commodity VLSI chip.

While the goal is to make the architecture generally programmable, there are some fixed attributes that cannot be avoided. In particular, the accelerator is based on DCT compression of 8x8 blocks. Support of other block sizes would require larger blocks of memory in both the commodity DCT VLSI and in the ASICs. In addition, the leading proposals for MPEG algorithms use an 8x8 block size. While the block size is fixed, the particular ordering of the blocks, or how the transformed blocks are used is totally programmable. The transformed blocks may be used to encode intensity or color information, they may be used in a 4-1-1 or 4-2-2 or 4-4-4 ratio, and may be used to transmit difference or absolute frame information.

The architecture will be presented in the context of decoding the CCITT P*64 standard, and it is assumed that the reader is familiar with this standard. Figure 1 shows the data flow for the decompression process of the P*64 decompression format. The diagram shows both software and hardware functions and indicates hardware functions by highlighting them in gray boxes.

The raw data comes in the top, and is placed in a data buffer by the I/O subsystem. A software process examines the raw data and extracts coefficient run data and header information for a whole frame, and places the result into two separate buffers. At the highest data rate, the software process must decode 640K Huffman encoded symbols per second. Preliminary estimates indicate that this would consume about 10% of the cycles of a single 40Mhz CPU.

The coefficient run data for an entire frame is picked up by the **DCT Unit**, which expands it, performs inverse quantization, and finally performs an inverse DCT. The resulting 8x8 blocks are then transfered to buffers in main memory. The maximum data rate out of the DCT unit is less than 4.5Mbytes/second, which is only 12-25% of the bandwidth available through commodity VLSI DCT chips.

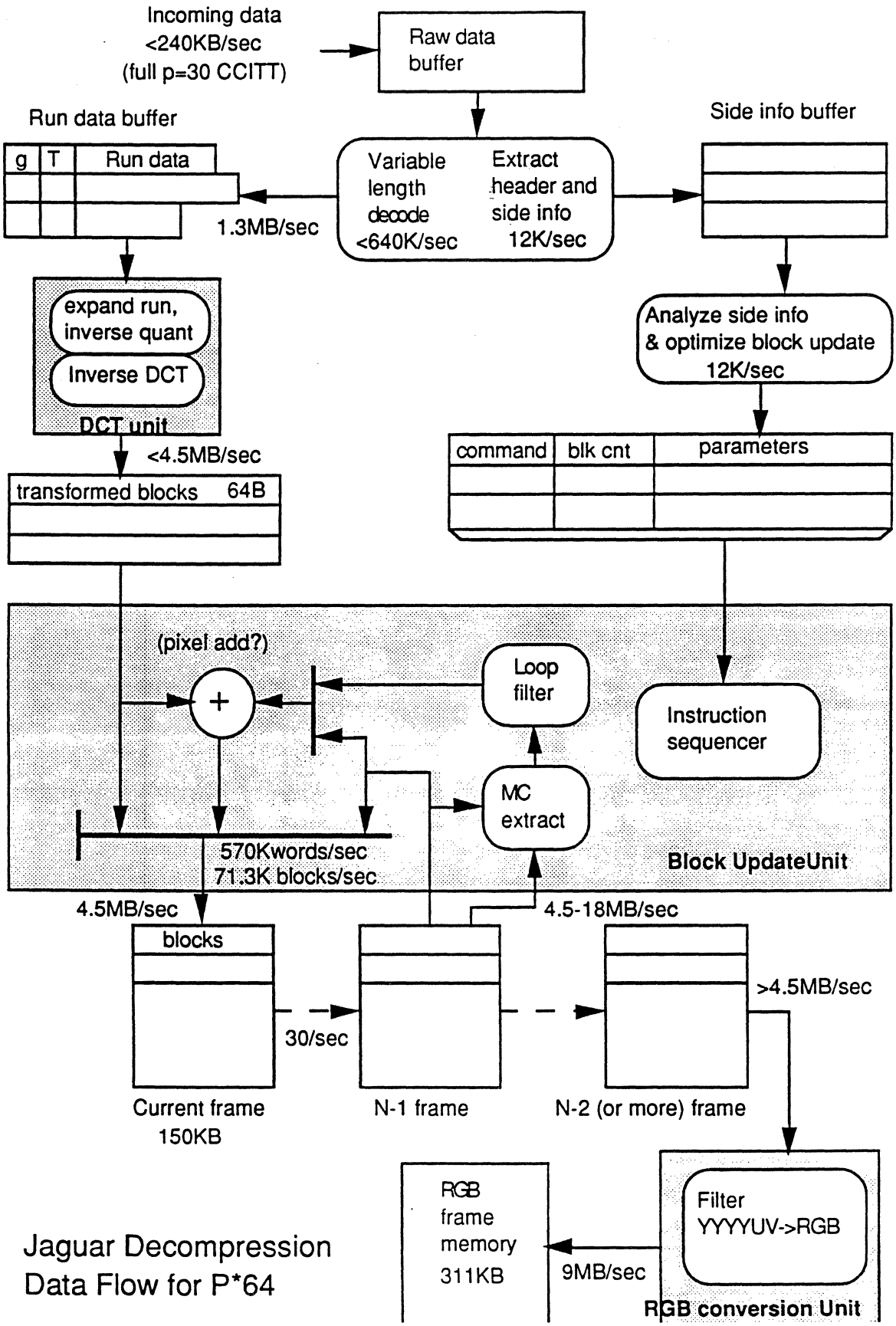
While the hardware is transforming the coefficient data, a second software process analyzes the header information for the frame that is contained in the side info buffer, and generates a command list for the **Block Update Unit**. This software process must decode and analyze on the order of 12K macro block headers per second.

The block update unit is a hardware unit that combines the previous frame with the transformed blocks to create the next frame in the sequence. It reads commands from the command list generated by the header analysis process. It can update a block of the current frame by copying a block from the transformed data or from the previous frame, or add the transformed block to the block of the previous frame. In addition, it can extract an off grid block from the previous frame for motion compensated blocks. The output of the block update unit is a completed fixed size frame in 8x8 YUV format blocks ordered in the sequence that they would be received.

The block update unit must process about 72K blocks per second. Each block is 64 bytes, which results in a maximum data rate of 4.5Mbytes/second. In the worst case, each block requires a command from the command list. Since the block update operations are really more data movement, and not so much computation, the 4.5Mbyte/second bandwidth through the hardware is easily achievable.

The frames are kept in a round robin buffer arrangement, that allows for some elasticity in the decompression rate. The last frame in the round robin buffer is picked up by the **RGB Conversion Unit** which converts it from YUV color space into RGB color space, and also generates a scan line ordered stream that is fed through a window DMA channel to its final destination. This unit may provide some post filtering to remove artifacts due to the block nature of the encoding schemes, and may also provide filtering for scaling.

One uncertainty here: it may not be possible for the RGB conversion unit to provide a scan line serial output and also be general in the mapping of its input blocks. If this not possible, then the output may not be scan line ordered, and an intermediate buffer may be required in order to update the screen with window DMA.


Incoming data
<240KB/sec
(full p=30 CCITT)
Raw data buffer
Run data buffer
g
T
Run data
1.3MB/sec
Variable length decode <640K/sec
Extract header and side info 12K/sec
Side info buffer
expand run, inverse quant
Inverse DCT
DCT unit
<4.5MB/sec
transformed blocks 64B
Analyze side info & optimize block update 12K/sec
command
blk cnt
parameters
(pixel add?)
+
Loop filter
Instruction sequencer
MC extract
570Kwords/sec
71.3K blocks/sec
Block UpdateUnit
4.5MB/sec
4.5-18MB/sec
blocks
30/sec
>4.5MB/sec
Current frame
150KB
N-1 frame
N-2 (or more) frame
RGB frame memory 311KB
9MB/sec
Filter YYYYUV->RGB
RGB conversion Unit
Jaguar Decompression
Data Flow for P*64

In general the data processed by the computational units is in the form of 64 byte blocks. Read and write requests to the processor's memory are performed in 64 byte blocks, which allows for a 70% efficiency of the processor's burst memory bandwidth. The final decompressed output is in the form of a window DMA stream that is passed through Wilson to get to the frame buffer.

In other cases, such as motion compensation, random access is required. Serial accesses tend to be more efficient than random accesses, since the memories may be left in page mode between accesses, but even the random block accesses are reasonably efficient, since a 64 byte block allows the memory system to stay in page mode for four consecutive cycles.

For operation of the pipeline as shown in figure 1, the memory bandwidth requirement is a little less than 50MB/sec in the worst case. The average requirement is less, since not all blocks require motion compensation, and not all blocks in a frame are coded, but simulation will have to be done to get a more accurate estimate of the average bandwidth.

Random	WDMA	Function
1.3M		Run data to DCT
4.5M		Transformed blocks out of DCT
4.5M		Transformed blocks into Block update
4.5-18M		Previous frame data (18M is worst case for motion compensation)
4.5M		Output of current frame
4.5M		Input to RGB converter
	9.0M	RGB converted data. (this is a serial Wilson stream)
24-38M	9M	Total bandwidth requirements

DCT unit

The DCT unit performs the most compute intensive operation in compression: the discrete cosine transform. The unit uses a commodity VLSI chip for the DCT computation.

The DCT unit is capable of performing five functions: 1) Huffman decoding of run data, 2) expansion of run data into 64 coefficients, 3) inverse quantization of coefficients, 4) arbitrary mapping of coefficient order into 8x8 blocks, and 5) forward or backward DCT. There are bypasses provided, so that software can perform any of these functions if desired. In the P*64 example described earlier, the Huffman decode is performed in software because P*64 requires that the coefficients be decoded in order to determine beginning of the next header. On the other hand, MPEG will probably provide a forward pointer to the next header, because this makes random access of frames easier.

In the worst case P*64 could require as much as 4.5Mbyte/sec output out of the DCT chip. Commodity parts have bandwidths of 20-40Mbyte/sec, which should provide sufficient bandwidth.

The next figure shows the logical data flow through the DCT unit.

The serial input stream is optionally passed through a programmable Huffman decoder. The Huffman decode tree is maintained in a small RAM that can be reprogrammed for different codes. While P*64 does not require hardware support for Huffman decoding, it is felt that MPEG will require higher bandwidth, and could possibly swamp the CPU.

The output of the Huffman decoder goes through the inverse quantization function that scales the level data to produce a 12 bit coefficient from 8 bit data. The scaling constants, g and T, are provided by the header decode process. The inverse quantization function is simply a multiply and add.

The level data is then passed with the corresponding run length, to the run expander that expands the single coefficient to a sequential stream of the length indicated by the run length value.

The resulting sequential stream must then be mapped into the 8x8 coefficient block. P*64 has a zigzag map that fills the 8x8 block in a zigzag pattern starting at the 0,0 point. This pattern is efficient, since most of the energy in coded blocks is close to the 0,0 point. As it turns out, a 4:2:2 chroma sample ratio would benefit more by a different zigzag pattern than the pattern used in a 4:1:1 sample ratio. For this reason, it is desirable to allow the zigzag pattern to be programmable, and to allow for a the selection of different tables for chroma blocks than for luminance blocks. The block mapper includes a loadable table that allows the zigzag pattern to be programmed.

The block buffer is double buffered, so that one set of coefficients can be assembled, while the previous set is being transmitted to the DCT VLSI. The DCT VLSI is a commodity part that performs an 8x8 inverse DCT. The output of the IDCT then is passed out to memory.

There are several bypasses that allow the input to the DCT unit to bypass the Huffman decode or inverse quantization logic. This allows software to perform these functions should some algorithms require operations not supported by the hardware.

The DCT unit reads commands from memory, and produces 64 byte blocks of expanded and transformed data. In some cases, the data is included in the data stream, in other cases, the data is pointed to by a data pointer.

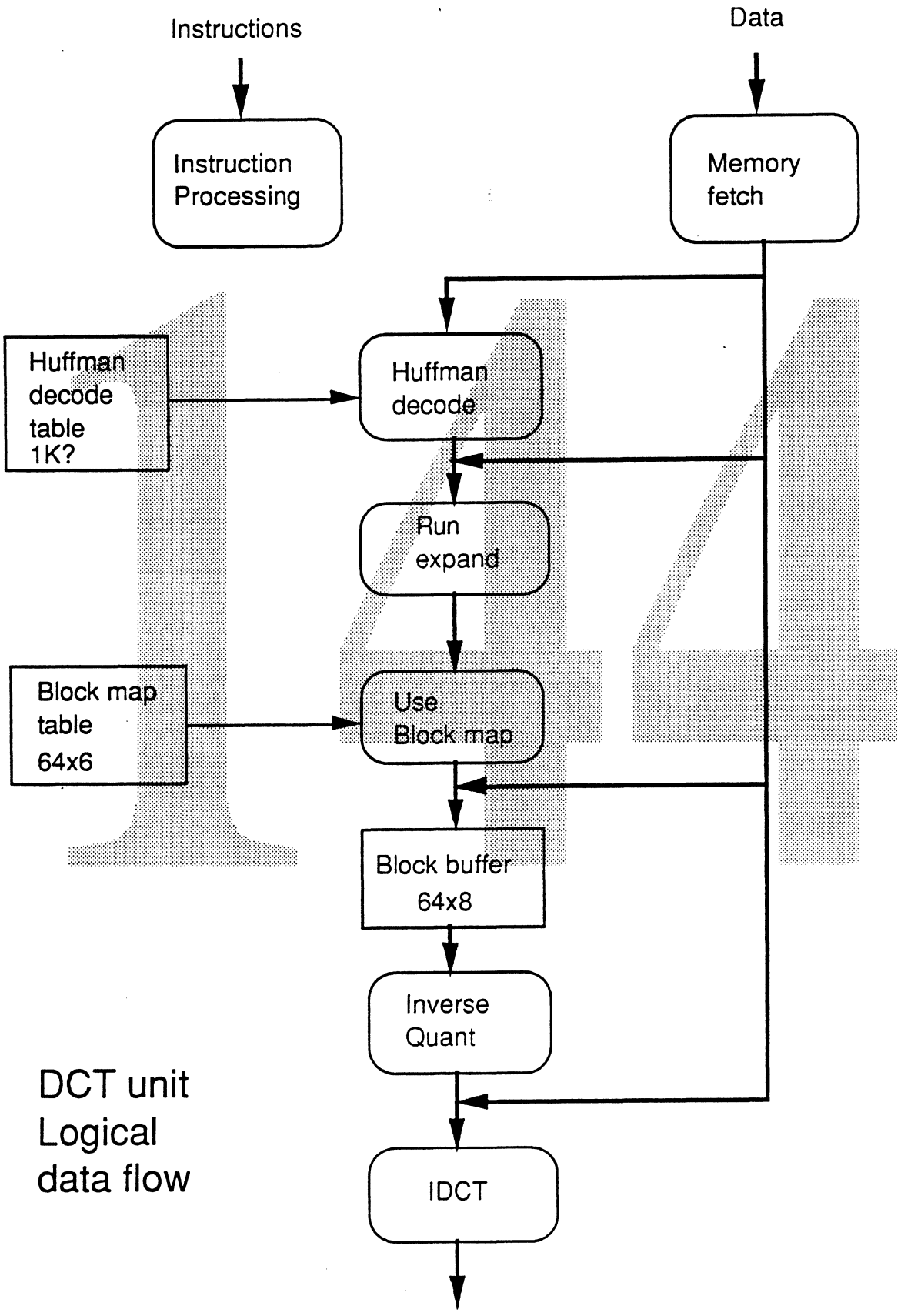
The DCTOpRunCodedBlock command indicates that the data following the Op code and length is run coded data that should be expanded and transformed to produce a transformed output block. Run coded data consists of a series of two byte *events*. The level field indicates the value of a nonzero coefficient, while the run field indicates the number of zero coefficients following the nonzero value.

The DCTOpHuffmanCodedBlock command requests that the Huffman coded data pointed to by the command be passed through the Huffman decoder before it is passed through the run code expander and transformed to produce an output block.

The DCTOpUseBlockMap modifier to the CodedBlock commands indicates that the block map ram should be used to remap the order of the coefficients. The block map is a 64 by six bit RAM that indicates the new coordinates (u, v) of sequentially recieved input samples. The BlockMap data type defines the format of this table. If this modifier is not used, the standard zigzag block map is used.

TheDCTOpQParms command sets the quantization parameters g and T that are to be used in the inverse quantization operation.

The DCTOpHalt command tells the DCT unit to halt.



DCT unit
Logical
data flow

Block update unit

The block update unit is shown in the middle highlighted block of figure 1. It merges the transformed blocks from the DCT unit with the previous frame to generate the next frame in the sequence.

The block update unit reads a command list that is generated by the header analysis software process. The unit supports three basic update operations: block replacement (from either transformed block or previous frame) delta block update, and motion compensated update.

The computation required for block update of non-motion compensated encodings is relatively small, and would probably take <20% of the CPU. Unfortunately, panning can cause almost every block in a frame to require motion compensation for many successive frames. This would totally swamp a software motion compensation algorithm. For this reason, the block update unit is a necessary part of the decompression accelerator.

The block update unit is contained in an ASIC. Its function is not compute intensive, rather its purpose is to move data around. There is no intelligence built into the BU unit about a specific coding format. Rather, this kind of intelligence is built into the software that generates the command list for the block update unit to execute.

The block update unit operates on 64 byte blocks. It merges inverse transformed blocks from the DCT unit with blocks from the reference frame to generate a new frame. There are six address pointers that keep track of the location of these blocks: DestPtr, DeltaPtr, RefPtr, RefXPtr, RefYPtr, RefXYPtr. DestPtr points to the location where the updated block should be placed in the current frame being computed. DeltaPtr points to the block from the DCT unit that will be merged with a block from the Reference frame. RefPtr points to the block in the reference frame that is at the same position as the block in the destination frame. RefXPtr, RefYPtr and RefXYPtr point to blocks that are adjacent to the Reference block. These blocks are required for extracting a motion compensated block from the reference frame.

There are six instructions for loading the six block address pointers. The pointers are incremented after they are used, so that they need not be reloaded on sequential accesses. For example, since the destination blocks are generated sequentially, DestPtr need be loaded only at the beginning of a frame. The format of these six instructions is defined by the BUPtrCommand structure type.

There are eight basic block update commands. There are four flavors of reference frame selection, and to each of these four operations, the transformed Delta block can optionally be added to the reference data. The resulting block is written to the address pointed to by DestPtr. DestPtr and RefPtr are each incremented on each operation. If addition of the Delta block is specified, then DeltaPtr is also incremented. A count can be specified that indicates the number of sequential blocks to perform the specified operation on.

The BUOpZeroRef command indicates that a value of Zero should be used for the reference block. This operation is useful to pass the Delta frame on unchanged.

The BUOpUseRef command indicates that the block pointed to by RefPtr should be used as the reference frame.

The BUOpUseRefMC command indicates that a motion vector compensated block should be extracted from the four blocks pointed to by RefPtr, RefXPtr, RefYPtr and RefXYPtr and used as the reference block. The motion vector can be loaded by the BUOpSetMCdeltas command.

The BUOpUseRefMCandLoopFilter operates like the BUOpUseRefMC command except that it additionally applies the loop filter to the extracted block. The filtered block is then used as the reference frame.

The BUOpAddDelta is a modifier to the previous opcodes that indicates that the block pointed to by DeltaPtr should be added to the reference frame to produce the destination frame. If this operation is specified, then the DeltaPtr is incremented, otherwise it is not.

The BUOpSetMCDeltas sets the X and Y offsets to be used when extracting a motion compensated block from the four reference blocks. The format of the command is specified by the BUSetMCCommand type. DX and DY are bytes and can range from +7 to -7.

DCT and Block update unit instruction sequencers

The DCT unit and Block Update unit share similar control architectures. Both units read commands from an input stream, and make internal registers accessible to the CPU.

The functional declarations at the end of this document indicate the internal state registers that are made available to the processor. Some of these registers, for example the quantization parameters g and T in the DCT unit, can be set both by instructions or directly by software. This dual accessibility allows for flexibility in control and visibility into the engines.

Each instruction sequencer has a 64 byte instruction buffer. In order to maintain efficient bandwidth utilization of the CPU memory, it is desirable to make cache line memory accesses. The 64 byte buffer can hold two cache lines, which allows for double buffer read operation using cache line size memory accesses.

The instruction pointers BUIP and DCTIP are byte pointers that indicate the current position of the instruction parse. When one half of an instruction buffer is empty, a memory request is initiated to refill it to the next sequential 32 bytes. The halves of the instruction buffer are direct mapped to memory, so a given memory address always maps to the same buffer.

The least significant byte of each unit's control word (DCTcntl and BUcntl) maintain status bits for the sequencer. The IBstate bits indicate whether each half of the instruction register is full or empty. There are two bits, one for each half.

The singleStep and halt bits in the control register are used to control execution. When the halt bit is high, the sequencer will stop executing instructions at the completion of the current command. When the singleStep bit is high, the sequencer will set the halt bit high at the completion of the current instruction, so that only a single instruction is executed if halt is reset. In this mode, the processor resets the halt bit to begin execution of an instruction, and can subsequently poll the halt bit to determine when the instruction has completed.

Because the instruction buffer is accessible to the processor, a useful immediate execution mode is available. This is accomplished by setting the instruction pointer to zero, writing the instruction packet into the first few bytes of the instruction buffer, setting the IBstate bits to indicate that the buffer is full, and finally setting single step mode high and halt low. The instruction will be immediately executed from the instruction buffer, and the sequencer will halt at it's completion. The processor can poll the halt bit to determine when the instruction has completed.

Immediate execution would be desirable when using the Huffman decoder on the P*64 format. In P*64, the end of the block can only be determined by performing a complete Huffman decode of all the preceding data. Since it is necessary to find the end of a Huffman coded block to find the next header, the Huffman decode must be done immediately, and the resulting position of the end of data must be read back by the software to continue processing the next header.

RGB conversion unit

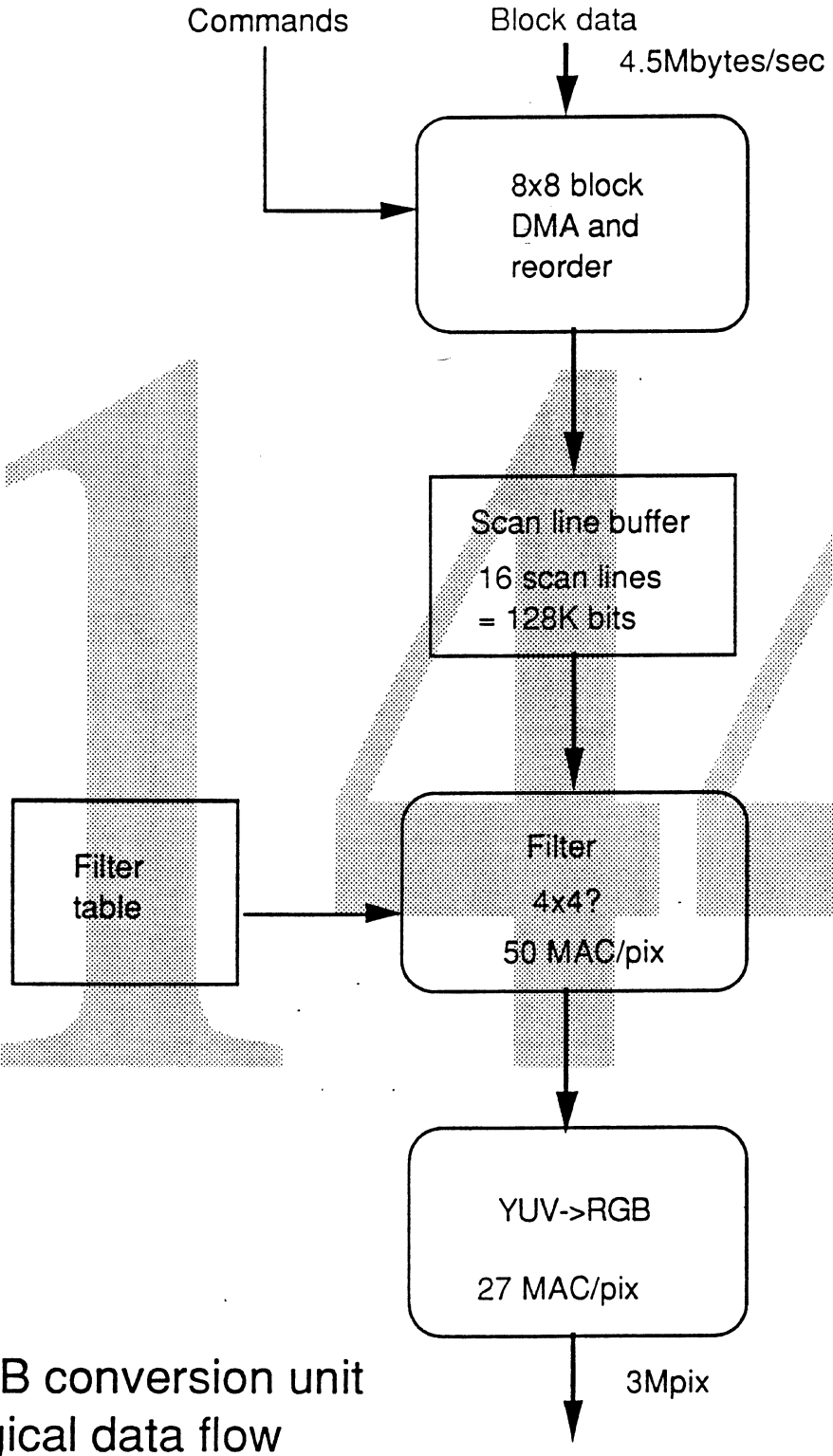
The description for this unit is still a little sketchy. In particular, the conversion from block to scan line requires a large memory of 128Kbits as it is described, and it may be more desirable to burn a little more system memory bandwidth than use a local memory of that size. This particular tradeoff is still being studied.

The RGB conversion unit reads in an 8x8 block organized frame in YUV color space, converts it to scan line order, filters the YUV components, and then converts YUV color space to RGB color space. The result is a scan line order RBG pixel stream that passed out through a window DMA channel to its final destination.

Figure 3 shows a logical data flow for the RGB conversion unit. The unit is controlled by an instruction stream that tells it the relative addresses of the blocks to read in. These blocks are read in and placed in a 16 scan line by 360 pixel buffer. Since the blocks are organized as 8 pixels by 8 scan lines, an entire 8 scan lines worth of blocks are needed before the topmost scan line of that group can be scanned out. The sixteen scan line buffer allows for overlapping the reading and outputting these 8 scan line groups. The scan line buffer would probably be implemented with an external SRAM, since it would require around 128K bits of storage.

The scan line data is read from the scan line buffer and filtered to produce a stream of scan line ordered YUV encoded pixels. The filtering is required to up sample the color difference components to the sample rate of the luminance channel. This filtering may also be useful for performing arbitrary scaling, but more work has to be done here before we include this functionality. The filtering is a 4x4 kernel that is implemented as two passes of a 4x1 kernel. The computational requirement turns out to be about 16 MAC's per pixel, so at the 3MPIX data rate of P*64, this is a 48MAC/sec computation rate.

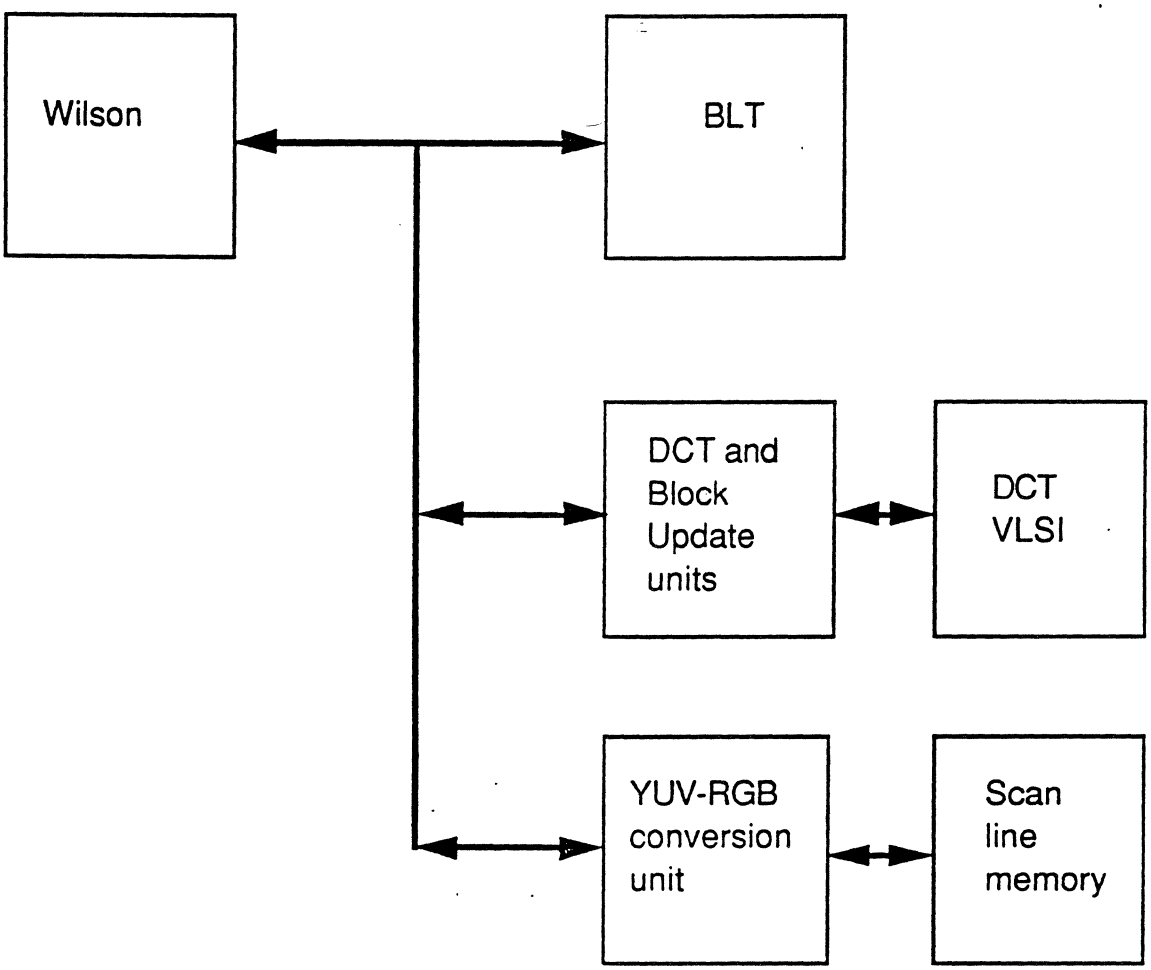
These pixels are then put through a YUV to RGB converter which is a implemented as a 3x3 matrix transformation from YUV coordinates to RGB coordinates. At the P*64 rate of 3Mpix, this requires 27MACs/second of computation.



RGB conversion unit
Logical data flow

Implementation

The most logical place for the decompression hardware to connect to the system is on the Wilson to BLT bus. This bus allows for both random accesses and sequential Window DMA accesses. The three hardware acceleration units are partitioned onto two ASICS that each interface to this bus.



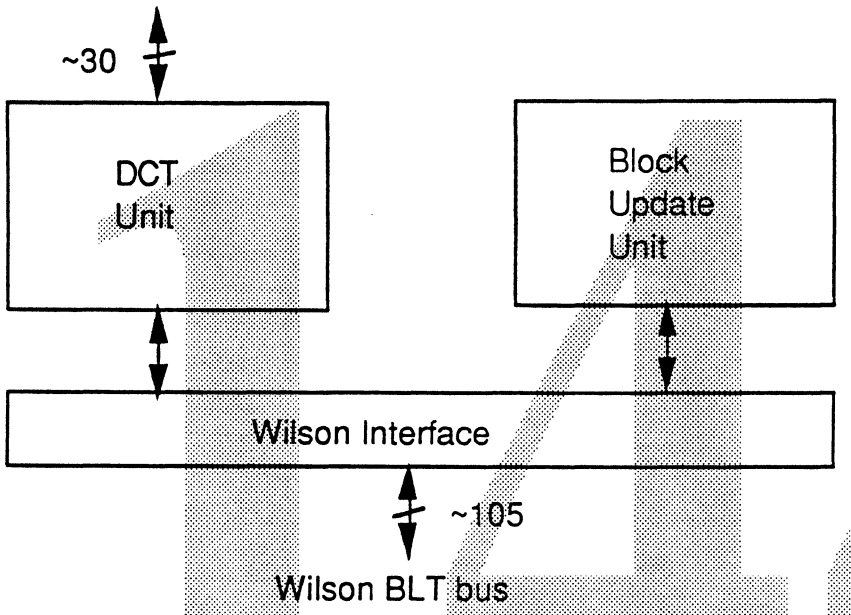
System Connection of Decompression Acceleration Units

The block update and DCT units are combined into one ASIC, the DCT/BU chip. The two units operate essentially autonomously. They each maintain independent buffering, memory sequencers and instruction processors. The two units do share a single bus interface. There is a connection between units that allows for the direct forwarding of transformed blocks from the DCT unit directly to the BU unit. This connection is useful in some configurations because it saves the memory bandwidth required to buffer the transformed blocks.

In addition to interfacing to the BLT bus, the DCT/BU chip is also connected to the commodity DCT VLSI chip that actually performs the computation for the inverse DCT.

The RGB conversion unit is on the second ASIC. Since RGB conversion requires a significant amount of computation, and will require a corresponding amount of silicon. This unit is not well defined at this time. At this time, it will probably require a 128K bit scan line store, that would probably be maintained in a RAM external to the chip.

There is still a possibility that the overall gate and pin count may be low enough that all three units could be put on a single ASIC, but it is to early to tell.

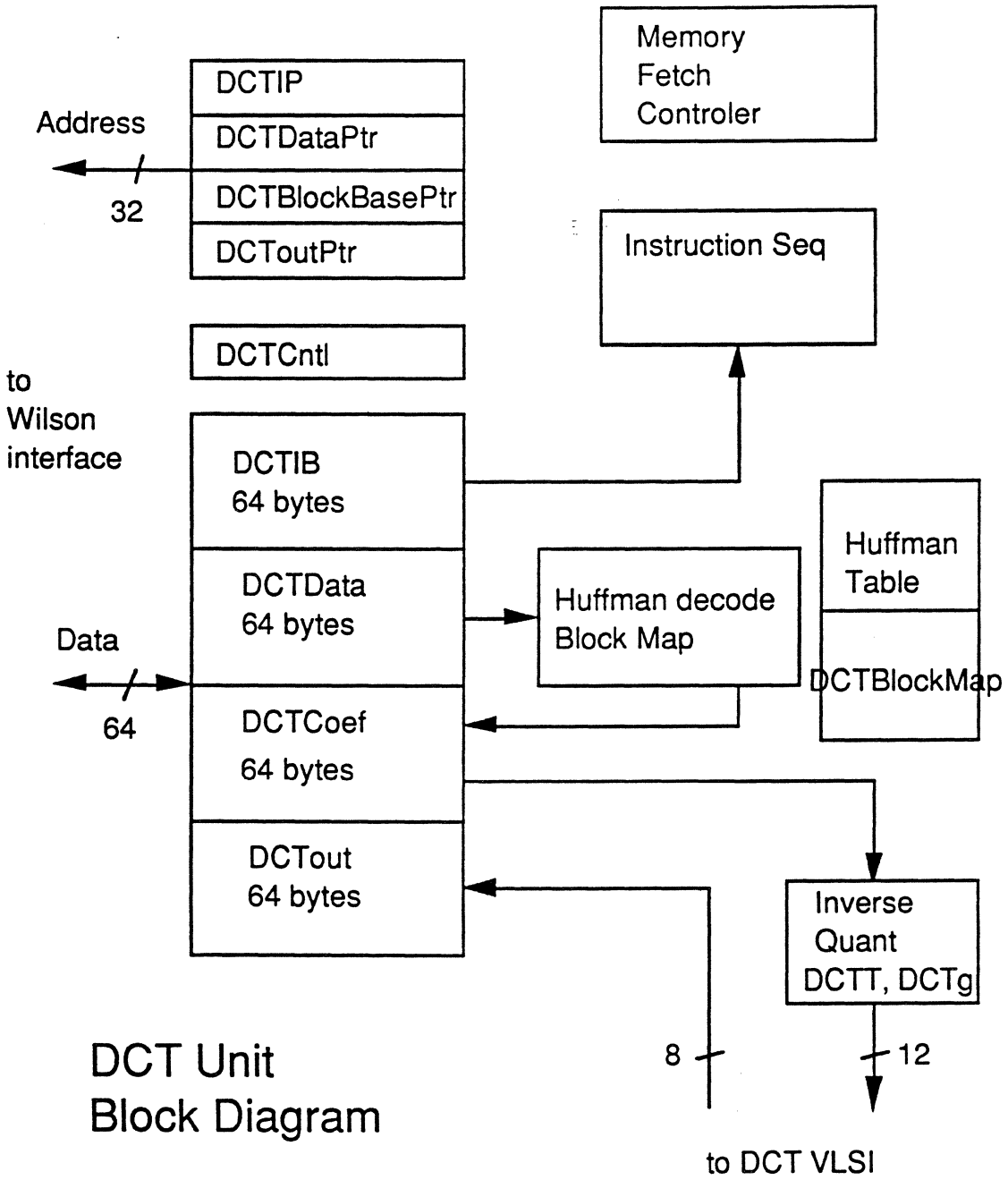


BU/DCT chip block diagram

The DCT unit and BU unit share a similar instruction sequencer and memory controller design. All memory data movement is performed into and out of a single port time multiplexed RAM block. The memory controller fetches and stores data to/from the addresses specified by a pool of memory pointers in 32 or 64 byte blocks. The memory controller also responsible for incrementing the memory address pointers. The instruction sequencer reads instructions form the instruction buffer in the RAM block, and sequences the data paths through the specified operations. The RAM block, memory pointers, and other state within the unit are directly readable and writable by the software, as well as by the operation of the instruction sequencer.

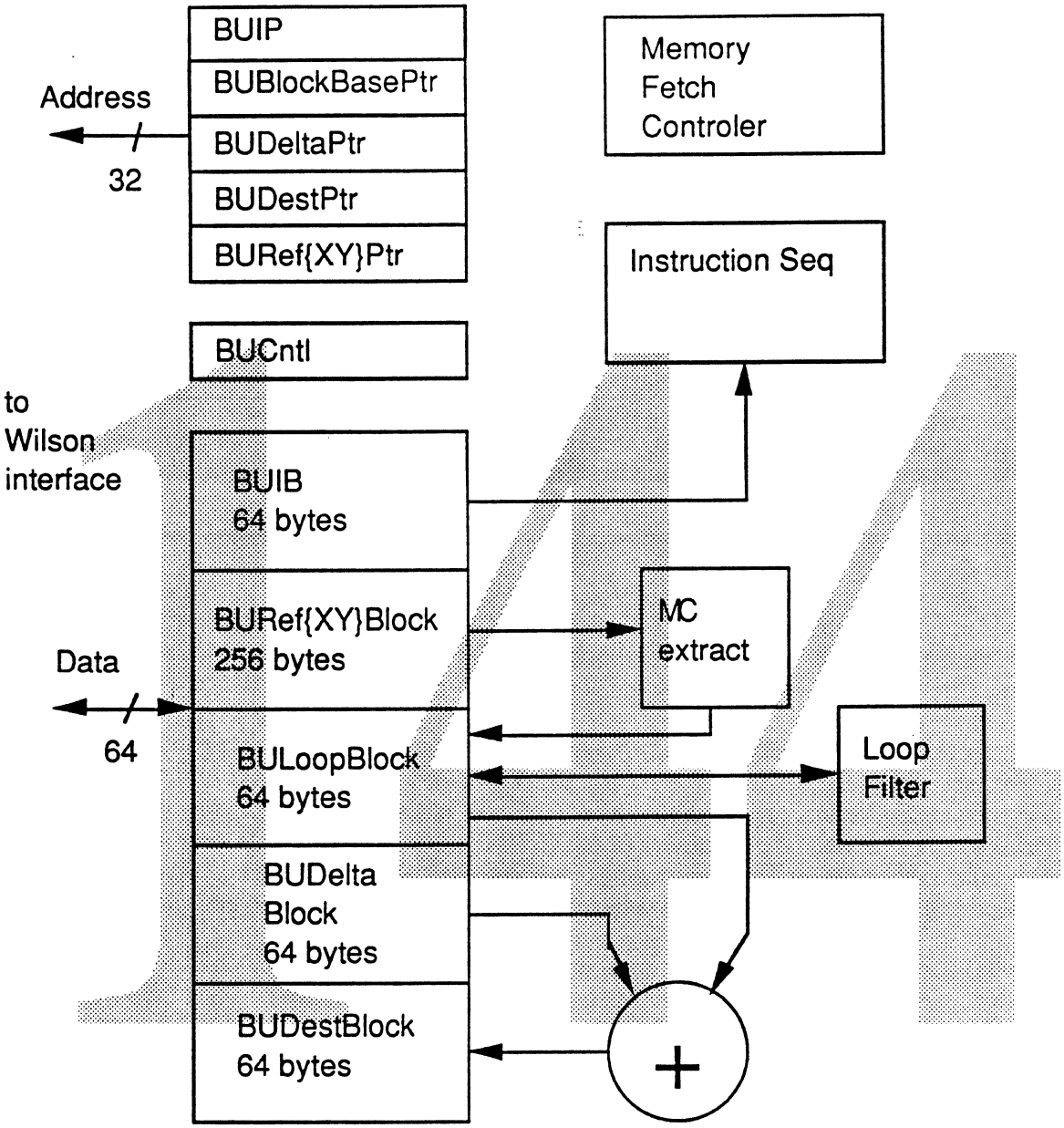
The DCT unit data paths include a Huffman decoder that decodes bit data into events. The decode table is maintained in a RAM that may physically be the same RAM block that is used to store data.

The DCT unit also includes a block map function with a loadable map that allows for remapping the order of events. The mapped data is placed in a coefficient buffer. After all the events for a single block are received, the block is sent to the commodity DCT VLSI chip. for transformation. The transformed data is returned and placed in a buffer in the shared RAM block, from which it is sent over the BLT bus to the processors memory.



DCT Unit
Block Diagram

The block update unit data paths include include a motion compensation extraction function that extracts an arbitrarily aligned 8x8 block of pixels from a 16x16 block of pixels. The extracted block can be optionally filtered by a loop filter that functions as specified by the P*64 standard. The filtered and motion compensated block can then be added to a delta coded block, as is required for frame differencing, and then written to memory.



Block Update Unit Block Diagram

At this time the RGB conversion unit is not well enough defined to give an implementation block diagram.

Sound compression

Since P*64 is designed to operate on top of ISDN in conjunction with a sound channel, the standard does not explicitly address sound compression. If Apple were to use P*64 for storage of compressed video, it we would have to address the compression and storage of high quality sound in along with video. In this case, there are a number of software algorithms that can be used for high quality sound compression with nominal CPU usage.

Since MPEG is targeted at the storage of high quality video, sound must also be compressed and stored with the compressed image. Since the sample rate of sound is much less than that of video, special hardware is not required for common sound decompression algorithms. Unfortunately, at this time we do not know what kind of sound compression will be standardized with MPEG, and there is a risk that the computational requirements for MPEG standard sound decompression could swamp the CPU. We will continue to track MPEG and make adjustments as necessary.

Issues

The architecture presented here is quite preliminary. We have not been working on the problem of decompression for very long, and the problems are neither completely understood or fully resolved. The major issues are as follows:

The Jaguar project has not yet begun to perform simulations on the compression algorithms that we are targeting. While the P*64 specification is reasonably understood, many parts of the specification are incomplete and require expert interpretation. We must validate our understanding with real simulations. In addition, the performance of CCITT P*64 at low bit rates is quite poor, and can be greatly enhanced by pre- and post-processing that is not specified by the standard. If we are to operate at low bit rates, we must understand and simulate this kind of pre- and post-processing.

The MPEG standard is just at the beginning of it's evolution. We are depending on expert vision to understand the range of the possible outcomes of this evolution, and we are targeting the hardware to handle this range. It is possible that we may not make the right choices here, and that we may not achieve complete MPEG compatibility.

We are still quite early in the design process of the hardware. We are just past the phase of understanding the bandwidth requirements and have made a first cut in partitioning the problem. The next level of detail is a more accurate sizing of the computational requirements on the software, and the gate counts on the hardware. We are at the point now that we can make a reasonable cut at the gate counts on the DCT/BU chip, but we are not at the same point for the RGB conversion unit. The risk here will be that the silicon costs will be higher than that required to meet the $50 cost target.

Additional wires are required on the E&W BLT interface to allow the decompression accelerator to share this interface with BLT. These wires are not defined at this time.

Software interface specifications

The following C declarations are very preliminary. They are software specifications for the data structures and instruction sets for the decompression hardware accelerators. Some functional simulation code for the hardware accelerators is also included for the purpose of clarifying their function. This code has not been debugged, and the definitions are still in flux, so don't expect to start writing production code with these.

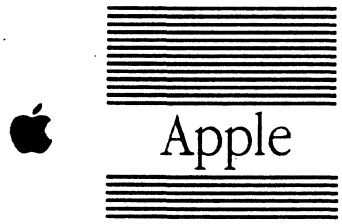
```
#define byte char

/* Data Structures for Jaguar Decompression Accelerator */

#define DCTBLOCKSIZE (8*8)
typedef byte Block[DCTBLOCKSIZE];

/* DCT unit Data structures */

#define DCTOpLoadDataPtr 0x20

typedef struct {
        byte OpCode;
        char bitPtr;
        char *dataPtr;
        } DCTOpLoadDataPtrCommand;

#define DCTOpUseBlockMap 0x8
#define DCTOpHuffmanCodedBlock 0x10
#define DCTOpQuantizedBlock 0x11
#define DCTOpUnCodedBlock 0x12

typedef struct {
        byte OpCode;
        char pad;
        } DCTOpHuffmanCodedBlockCommand;

#define DCTOpRunCodedBlock 0x13

typedef struct {
        byte run;       /* Run is actually 6 bit field. Spare two bits should be high order bits. */
        byte level;
        } Event;

typedef struct {
        byte OpCode;
        byte length;
        Event data[];
        } DCTOpRunCodedBlockCommand;

#define DCTOpQParms 0x31
```

```
typedef struct {
        byte OpCode;
        byte pad;
        int g;
        int T;
        } DCTOpQParmsCommand;

#define DCTOpHalt 0
#define DCTOpNop  1

typedef struct {
        byte OpCode;
        byte pad;
        } DCTOpCntlCommand;

/* DCT unit software visible state */

struct {
        unsigned pad : 2;
        unsigned x : 3;
        unsigned y : 3;
        } DCTBlockMap[64];

struct {
        unsigned long reserved : 28;
        unsigned IBstate : 2;     /* How full is instruction buffer? */
        unsigned singleStep : 1;
        unsigned halt : 1;
        } DCTcntl;

char DCTInstructionBuffer[64];

char *DCTIP;              /* Instruction pointer */
char *DCTBlockBasePtr;
Block *DCToutPtr;
int DCTg, DCTT;                   /* Quantization parameters */
char DCTBitOffset;        /* Used for Huffman data.  Only three bits are used here. */
char *DCTDataPtr;         /* Coded Data pointer.  Huffman data also needs BitOffset. */

/* DCT Unit functional simulation */

void DCTUnitExecute()
{
}

/* Block Update Unit Instruction Set */

/* Load one of six address registers in Block Update Unit.  Addresses are Block addresses. */
```

```
#define BUOpLoadDeltaPtr      0x10    /* pointer to transformed delta blocks */
#define BUOpLoadRefPtr                0x14    /* pointer to Reference block */
#define BUOpLoadRefXPtr               0x15    /* pointer to adjacent ref block in X direction for
MC */
#define BUOpLoadRefYPtr               0x16    /* pointer to adjacent ref block in Y direction for
MC */
#define BUOpLoadRefXYPtr      0x17    /* pointer to diagonal ref block for MC */
#define BUOpLoadDestPtr               0x11    /* pointer to destination address */

typedef struct {
        unsigned OpCode : 8;
        unsigned long Address : 24;
        } BUAddressCommand;

/* Block Update operations */

#define BUOpZeroRef           0x20            /* Use zero as reference block */
#define BUOpUseRef            0x21            /* Use reference block */
#define BUOpUseRefMC          0x22            /* Use motion compensated reference block */
#define BUOpUseRefMCandLoopFilter 0x23    /* use motion comp reference block and loop filter
*/
#define BUOpAddDelta 0x10             /* Modifier to previous opcodes.  Add delta block to ref. */

typedef struct {
        byte Opcode;
        byte Count;
        } BUBlockCommand;

/* Set motion compensation vector */

#define BUOpSetMCdeltas       0x40

typedef struct {
        byte OpCode;
        byte pad;
        char DX;
        char DY;
        } BUSetMCCommand;

/* Random control commands */

#define BUOpNop                       1
#define BUOpHalt              0
#define BUOpSwapMCX                   0x50    /* Swap reference blocks around X axis */
#define BUOpSwapMCY                   0x51    /* Swap reference blocks around Y axis */

typedef struct {
        byte OpCode;
```

```
        byte pad;
        } BUCntlCommand;

/* Block Update Software visible state. */

Byte *BUBlockBasePtr;           /* Base pointer to CPU memory that contains blocks. */
char *BUIP;                             /* Block Update Unit Program Counter */
unsigned long DeltaPtr, DestPtr, RefPtr, RefXPtr, RefYPtr, RefXYPtr;
Block RefBlock;                         /* Chip internal block buffers, 64 bytes each */
Block RefXBlock;
Block RefYBlock;
Block RefXYBlock;
Block DeltaBlock;
Block LoopBlock;

struct {
        unsigned reserved1 : 8;
        unsigned
        unsigned swapX : 1;             /* swap X state */
        unsigned shiftX : 3;        /* MC vector */
        unsigned swapY : 1;             /* swap Y state */
        unsigned shiftY : 3;        /* MC vector */
        unsigned reserved2 : 4;
        unsigned IBstate: 2;        /* how full is instruction buffer? */
        unsigned singleStep : 1;
        unsigned halt : 1;              /* Halt/run state */
        } BUCntl;

char BUInstructionBuffer[64];

/* Block Update Functional Code */

#define get3bytes (((long) *BUIP++) << 16 | ((long) *BUIP++) << 8 | *BUIP++)
#define BlockMemoryAddress(p) (BUBlockBasePtr + DCTBLOCKSIZE*(p))
Block ZeroBlock = 0;

void BlockUpdateUnitExecute()
{
char OpCode;
int count;

        while (!BUCntl.halt) switch (OpCode = *BUIP++) {

        case BUOpHalt: BUCntl.halt = 1; break;
        case BUOpNop: break;


        case BUOpLoadDeltaPtr:          DeltaPtr = get3bytes; break;
        case BUOpLoadRefPtr:  RefPtr = get3bytes; break;
```

```
        case BUOpLoadRefXPtr:       RefXPtr = get3bytes; break;
        case BUOpLoadRefYPtr:       RefYPtr = get3bytes; break;
        case BUOpLoadRefXYPtr:      RefXYPtr = get3bytes; break;
        case BUOpLoadDestPtr:       DestPtr = get3bytes; break;

        case BUOpZeroRef:
        case BUOpZeroRef | BUOpAddDelta:
        case BUOpUseRef:
        case BUOpUseRef | BUOpAddDelta:

                for (count = *BUIP++; count >0; count--) switch(OpCode) {

                case BUOpZeroRef:
                        AddBlock(ZeroBlock, ZeroBlock, BlockMemoryAddress(DestPtr++));
                        RefPtr++;
                        break;

                case BUOpZeroRef | BUOpAddDelta:
                        AddBlock(ZeroBlock,
                                        BlockMemoryAddress(DeltaPtr++),
                                        BlockMemoryAddress(DestPtr++));
                        RefPtr++;
                        break;

                case BUOpUseRef:
                        MoveBlock(BlockMemoryAddress(RefPtr++), RefBlock);
                        AddBlock(RefBlock,
                                        ZeroBlock,
                                        BlockMemoryAddress(DestPtr++));
                        break;

                case BUOpUseRef | BUOpAddDelta:
                        MoveBlock(BlockMemoryAddress(RefPtr++), RefBlock);
                        AddBlock(RefBlock,
                                        BlockMemoryAddress(DeltaPtr++),
                                        BlockMemoryAddress(DestPtr++));
                        break;
                        }

        /* Motion Compensation is not implemented yet. */
        default:
                printf("BlockUpdate: unimplemented or bad OpCode: %x.\n", OpCode);
                }
}
```



Jag1 Expansion Interface & Wilson

External Reference Specification
Special Projects
Jaguar

November 12, 1989
Version 1.0

Return Comments to:
Dean Drako
Phone: x4-4497
AppleLink: DRAKO
MS: 60D

144

Apple CONFIDENTIAL

Contents

Section 1

Introduction

This document is an external reference specification for the Jag1 implementation of the Expansion I/O Interface and the Wilson Architecture. It discusses issues surrounding the hardware interface to the Jaguar Expansion I/O (BLT), describes the Expansion Interface and Wilson Chip (E&WC), describes the functionality of Wilson, and discusses some of the high and low level software issues relating to Wilson.

This document is not complete. It is the first attempt at describing the implementation. Any and all input would be greatly appreciated.

There are several sections to this document. Section 2 gives an overview of the Wilson Architecture and its goals in Jag1. It also provides an outline summary of the functionality. Section 3 and Section 4 discuss the software needed to attain these goals and the software needed to support the Wilson hardware. Section 5 details the hardware implementation of the E&W Chip. Section 6 provides example Wilson operations and sequences. Section 7 highlights outstanding issues.

What is Wilson?

Current CPU's and those of the foreseeable future will be unable to move data at the rates needed for live video input, video decompression/compression, or animation while still performing other tasks. Specialized hardware can be developed for these tasks which is simpler and has the needed capabilities. Some of this hardware has already been developed by third parties for Macintosh video input.

The Wilson Architecture is an attempt to standardize hardware for moving pixel regions. This will guarantee that all of the hardware can communicate. Wilson does not attempt to standardize the implementation of the hardware. It only attempts to standardize the fundamental data formats, a minimum amount of functionality, and the transfer mechanisms.

Wilson is an architecture for moving data between hardware devices. Its primary focus is the movement of regions from a pixel source to a pixel destination. Wilson attempts to solve some of the problems associated with many different sources (video input hardware, video decompression hardware, main memory, and CPUs) attempting to send information to many different destinations (frame buffers, main memory, and video compression hardware). This is a fundamentally different environment than the Macintosh. The Macintosh environment only supports a single source (the CPU). The source and destination devices may be built by Apple or third parties. Wilson attempts to guarantee that all third party hardware will communicate with all Apple hardware and all other third

party hardware. What does this mean to the user? It means he buy any manufacturer's video hardware (be it a frame buffer, a video input card, a compression card, a video effects card, or a graphics accelerator) take it home, plug it in, and it works with all of the other hardware he owns.

The Wilson Architecture provides for data movement, primarily pixel data movement. The data movement facilities can be utilized to support animation or other high bandwidth operations. The CPU is relieved of the data movement tasks and can therefore dedicate its processing power to generating the animation frames.

Clarification of Terms

The Wilson Hardware Architecture is the definition of the communications protocol used by the hardware. This is the definition of Wilson which a third party card developer would see. The Wilson Hardware Architecture is also referred to as the Wilson Architecture and simply as Wilson. The Wilson Software Architecture is the definition of Wilson that a toolbox or an application might see. This is potentially very different than the definition actually implemented in the hardware. It may include features which are implemented only on the motherboard and never really observed by third party hardware developers. The E&WC is the implementation of the Wilson Hardware Architecture on the motherboard of the Jag1 CPU.

The Wilson Documents

This is one of four documents which discusses Wilson. The four documents are:

1) Wilson -- Introduction (provides an overview of goals and issues)

2) Wilson Architecture Specification (details the architecture -- to be revised)

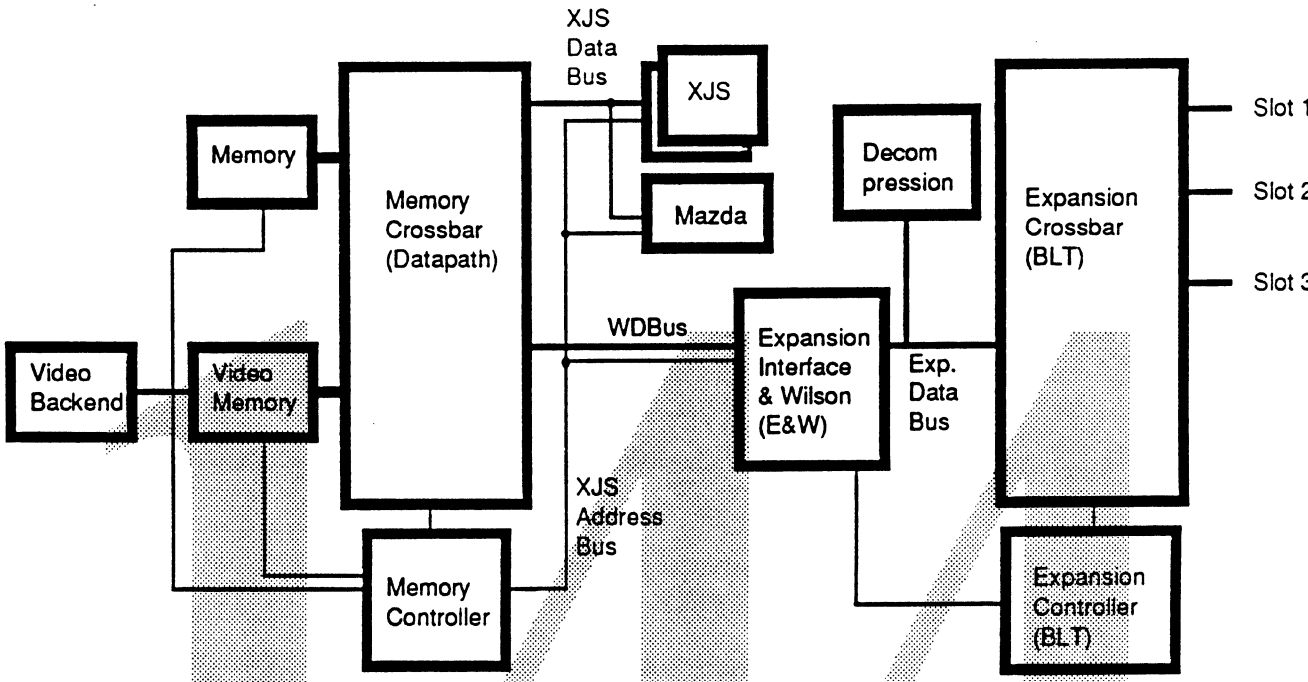
3) Wilson Architecture Specification Notes (parallels the Wilson Specification and provides some justification -- to be completed)

4) Jag1 Expansion Interface & Wilson ERS (details the implementation for the first Jaguar)

This document, the Jag1 Expansion Interface & Wilson ERS, describes the first implementation of the Wilson Architecture. It also describes the implementation of the Expansion I/O Interface for Jag1.

Jag1 Implementation of Wilson

A system block diagram may make some of the following information more easily understood. This diagram will be discussed fully in the hardware section of this ERS.



System Diagram Showing Location of Expansion Interface & Wilson Chip

The Expansion Interface and Wilson Chip sits between the Expansion I/O Crossbar (BLT) and the Memory Crossbar. It acts as a bus converter for normal transactions between the CPU's and the Expansion I/O. In addition the E&WC also implements the Wilson Architecture and related hardware for the motherboard of Jag1.

Definition of Terms

Alpha Stream -- A stream which contains only alpha information.

Bidirectional Device -- A device which has at least one Source Resource and one Destination Resource.

Channel -- Many different types of channels exist including Source Resource Channels, Destination Resource Channels, and Sequencer Channels. A Resource usually is capable of multiplexing its operations between a number of logical channels. Each of these channels has corresponding state information allowing the Resource to switch between them quickly.

Clip Alpha -- Same as a normal alpha value (basically transparency) except that a value of zero also indicates that the pixel should be clipped.

Destination Device -- A device which includes at least one Destination Resource.

Destination Resource -- Conceptual element which receives a stream of data from a Source Resource. Generally a Destination Resource contains information saying what it should do with the incoming stream.

Destination Resource Channel (DRC) -- A Channel which acts as the destination of a stream.

Device -- A hardware element in the Jaguar System (motherboard, expansion card, etc....). In the case of Wilson a device can either be a source of streams, a destination of streams, or both.

Expansion Interface & Wilson Chip (E&WC) -- The chip in Jag1 which implements the Wilson Hardware for the motherboard, some stream processing resources, and the Expansion I/O interface.

Pixel Stream -- A stream which contains both pixel intensity information and alpha information.

Port -- All Stream Processing Resources and Destination Resources have a Port. The port is the location where input data is received.

Sequencer -- The Sequencer can execute a simple instruction stream of move operations. It is used to load up the contents of the Source Resource and Destination Resource registers and to start their operation.

Sequencer Channel -- The sequencer can have several channels which operate independently using the same basic hardware.

Source Device -- A device which contains at least one Source Resource.

Source Resource -- Conceptual element which sends a stream of data to a Destination Resource. Generally the Source Resource contains registers to describe how the stream is created and where it should be sent

Source Resource Channel (SRC) -- A Channel which can source a stream. Generally located in a Source Resource.

Streams -- A sequence of bytes transferred from a source device to a destination device without any corresponding address information.

Stream Processing Resources -- A Resource which accepts streams as input, processes them, and sends out streams. It can generally be thought of as a data flow element.

Stripped Pixel Stream -- A stream which contains only pixel intensity information (no alpha).

Wilson -- Hardware Architecture for moving image data. This is the architecture that a third party hardware developer would see.

Wilson Architecture -- Same as Wilson.

Wilson Hardware Architecture -- Same as Wilson.

Wilson Software Architecture -- Not completely defined, but most likely the software definition which is implemented to abstract the Wilson Hardware. This may include some default Stream Processing Resources (which are not part of the Wilson Hardware Architecture).

Section 2

Concepts and Facilities

The Concepts and Facilities Chapter gives an overview of some of the new functions which we will attempt to implement in Jag1 with Wilson. Wilson is an attempt to define an architecture for moving data at rates beyond the capabilities of conventional CPUs. The Wilson architecture focuses primarily on handling data movement for compression/decompression of video, live video, and animation.

The Jaguar family of machines focuses on 3 new areas: 1) simulation, 2) media rich environments (text, graphics, video, audio), and 3) geographically separate people/data. To address these areas the Jaguar family will have certain characteristic: 1) interactive 2 and 3D operations, 2) time critical functionality, and 3) connections to the world.[1] Wilson and the Expansion I/O assist in attaining the first two goals on both lists.

Wilson Operations or Applications

This Wilson hardware and Wilson Architecture provide the operations listed below:

1) Live Video. A live video (NTSC or PAL) input device which fits in the Wilson Architecture is located on the Jaguar Expansion I/O. It can directly update a frame buffer region at 60 (or less) frames per second. The Wilson Architecture is used to transfer the data from the video input device to the frame buffer. The input device performs interpolation to convert from an interlaced to a non-interlaced signal. Two forms of video input device are envisioned: 1) minimum buffer version (low cost, 1 or 2 scan line buffers worth of memory) and 2) full back buffer version (provides for tear free updates & frame rate conversion). A method of beam chasing to avoid frame tears is used in the full back buffer version. The user can drag the video window from monitor to monitor without restrictions. The Wilson Architecture provides for conversion from 8 BPP to 24 BPP or 24 BPP to 8 BPP as needed to guarantee this.

2) Decompressed Video. A live video window is generated on the screen by a decompression engine. All the features of Live Video listed above are supported. Wilson can also be used to deliver the compressed image data to the decompression engine.

---

[1] Farrand, Toby, Hugh Martin, "Jag Mission," Internal Memo, Apple Computer, May 21, 1989.

3) Back buffered animation. All the animation is generated by the CPU in a main memory back buffer. The frame is transferred to the Frame Buffer using the Wilson Architecture. The E&WC performs the actual data movement. Beam chasing is used to avoid frame tears. The back buffer and the Z-buffer in main memory are cleared by the E&WC. The transfers and the clears are performed in sequence with the application.

4) Scrolling. The E&WC provides support for smooth scrolling of a full page. The scrolling can be accomplished in place or from a backbuffer. This is simply several large Wilson transfers performed by E&WC at the application's/toolbox's request.

5) Dynamic Objects. Wilson can be used to assist handling dynamic objects such as windows and icons. Wilson can be used to support the dragging of active windows and icons. It is conceivable that Wilson be used for cursor support, but this is not a design goal. The cursor will be supported by the Video Backend (Elmer) and/or by software.

6) Data Streams. Wilson can be used to transfer data streams to and from peripheral devices and local memory. These data streams are necessary for support of graphics accelerators and compression/decompression hardware.

7) Compositing. The E&WC provides limited compositing capabilities for image streams. This is primarily intended for repetitive real time compositing (placing antialiased text over live video for example). The hardware might be used to accelerate general compositing operations. The compositing operations are not directly part of the Wilson Architecture but are implemented by the E&WC. They fit within the architecture but are not part of the architecture.

8) Graphics Accelerators. The Wilson Architecture provides a method for 3D graphics accelerators or other image processing devices to process data and send it to a destination (frame buffer for example).

Wilson Architecture

The Wilson Architecture is described in detail in the Wilson Architecture Specification document. The fundamental ideas are summarized here. Wilson's primary focus is the movement of graphic & video image data.

There are two aspects to the Wilson Architecture: 1) Hardware Architecture and 2) Software Architecture. The hardware architecture applies to the motherboard's hardware and any third party hardware. Th e data types available in hardware are severely restricted to simplify the hardware. The data types available in software do not need to be so severely restricted. The following sections apply only to the Wilson Hardware Architecture. The Software Architecture is detailed in later sections.

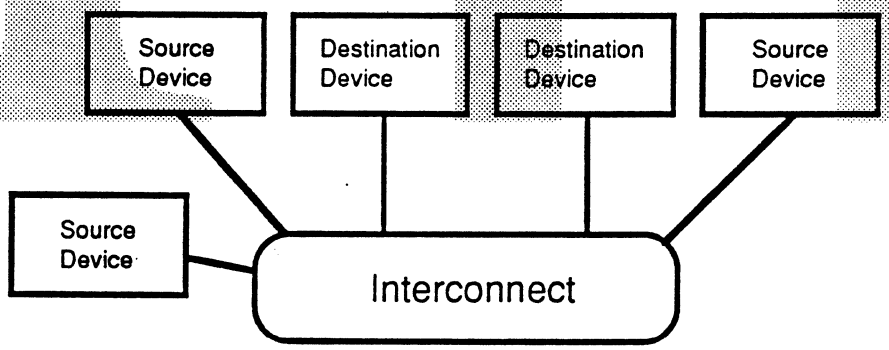
Multiple Sources and Destinations

The current Macintosh hardware and software model is based on a single source of video information. Every pixel displayed on every monitor is placed into the frame buffer by the CPU (almost always by Quickdraw). This model is not acceptable when one desires to add video and animation to the system. The CPU cannot place every pixel of video into the framebuffer. It simply does not have the bandwidth. The Wilson model of the video display system attempts to provide for multiple sources to display information on multiple destinations in a general fashion. The sources may be video input, video decompression, CPUs, Graphics Accelerators, or other new devices. The destinations are usually frame buffers located on the expansion I/O or motherboard but might also be video compression hardware or other new devices.

The sources in the Wilson environment can be thought of as intelligent DMA devices. They are capable of sending data to a destination when given the appropriate command(s). The overall concepts behind Wilson, however, extend far beyond ordinary DMA.

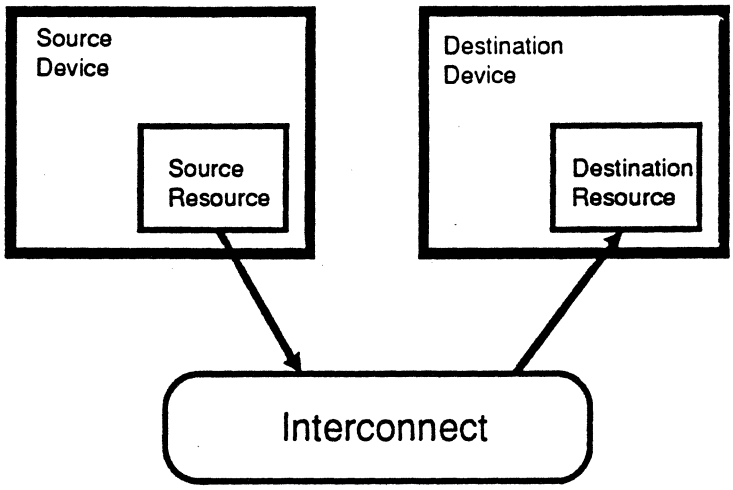
Source Resources and Destination Resources

The Wilson Architecture separates the source and the destination of pixel data. Devices are physical entities in the system. For example each expansion card in the system might be a device, main memory is a device, and video de-compression hardware would be a device. The separation of source and destination can be visualized as shown below.



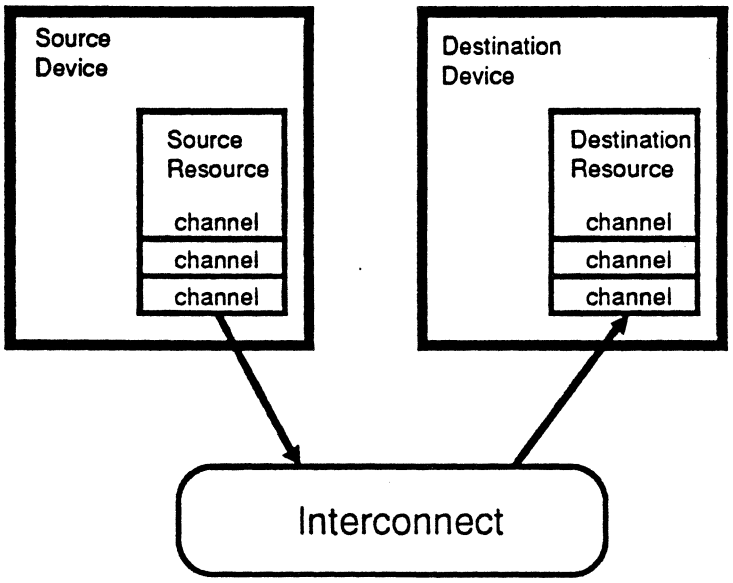
Multiple Sources and Destinations Connected Together

Each Source contains a Source Resource which is responsible for sending data from the source device. Each destination device contains a Destination Resource which is responsible for receiving data from Source Resources. A Source Resource and a Destination Resource are required to communicate in the Wilson environment. This is shown below.



Source Resource and Destination Resource in their Respective Devices

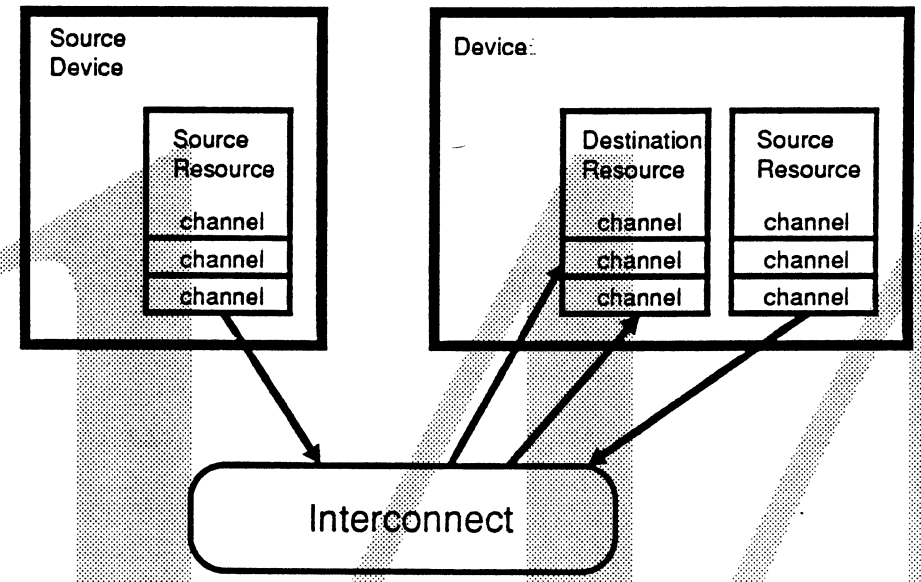
A Source Device may desire to send data to multiple destinations simultaneously. To do this it requires multiple Channels in its Source Resource. Each Source Resource Channel (SRC) functions independently to send data to a Destination Resource. The same is true of a Destination Device. It must have multiple Destination Resource Channels (DRC) if it desires to receive data from multiple Source Resource Channels simultaneously. A Source Resource Channel sending data to a Destination Resource Channel is shown below.



Source Resource and Destination Resource with Multiple Channels

A device may function as both a source and destination. In this case it is simply a "device" which includes both a Source Resource and a Destination Resource (each which may have multiple channels). A good example of a device such as this is main memory. An example is shown below. In

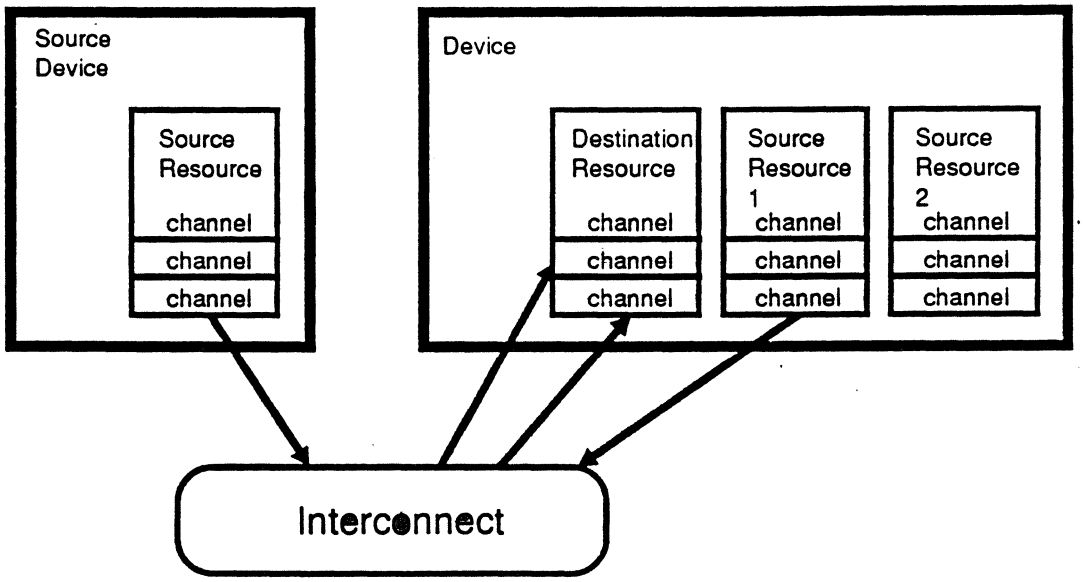
this example the Source Resource Channel in the Source Device is sending data to a Destination Resource Channel in our Bidirectional Device. The Bidirectional Device is using its Source Resource Channel to send data to one of its own Destination Resource Channels. Nothing in the Wilson Architecture prevents this and it is very useful for operations such as bit-blits.



A Source Device and a Bidirectional Device

It should be noted that each device in the system is different. This implies that each Destination Resource and Source Resource in the system will be slightly different. The function performed by each of these Resources is basically the same. The methods used, however, depend on the device under question. For this reason Source Resources and Destination Resources are specialized. For example if the device is main memory it will have a Memory Source Resource and a Memory Destination Resource.

It is conceivable that a device might desire different interface formats. It could therefore define two different resources for access. This is shown below.



A Resource with Multiple Source Resources

The two Source Resources provide access to the same data but the interface formats are different. This same concept could be used for multiple Destination Resources on the same device.

The Wilson Architecture attempts to guarantee that all Source Resources and all Destination Resources can directly communicate. It does this using Streams. Streams are explained in the next section.

Streams

The Wilson Architecture separates the source and the destination of pixel data. The source and the destination communicate via byte streams. A device can either source or receive byte streams. A stream does not contain the address of each pixel. It is simply a sequence of bytes transferred from the source to the destination. Each byte transferred is labeled with the stream number so that the destination knows what to do with the data. A Source Resource Channel (SRC) is responsible for creating a byte stream and a Destination Resource Channel (DRC) is responsible for accepting a byte stream.

The Source Resource Channel (SRC) is given the information describing how the stream should be created (location, x coordinate, y coordinate, size, width, length, format.... whatever the source designer feels is needed). The Destination Resource Channel (DRC) is given information describing how the data stream should be handled when it is received (where to put the data). The Source and Destination Resource Channel description information is set up by a CPU in the system. After the SRC and DRC are set up the CPU starts the SRC. The SRC creates the stream and sends it to the DRC.

Four common formats of streams, in addition to simple byte data, are defined to facilitate hardware communication:

1) 16 Bits per Pixel (8 bits grayscale & 8 bits alpha) called `k16_GRAY`

2) 32 Bits per Pixel (8/8/8 bits RGB & 8 bits alpha) called `k32_RGB`

3) 32 Bits per Pixel (16 bits grayscale & 16 bits alpha) called `k32_GRAY`

4) 64 Bits per Pixel (16/16/16 bits RGB & 16 bits alpha) called `k64_RGB`

The first two formats are used for normal grayscale and color communication in the system. Format 3 & 4 are defined so no one can call us short sighted. Smaller forms (1, 2, and 4 bits per pixel) will not be implemented in hardware. They will probably be supported in software. All graphics sources (video decompression, video input, graphics accelerators) must support at least one of the four data types. All graphics destinations (frame buffers, compression engines, back buffers) should support all four data types. Note that all four data types include a full alpha channel.

Specifying a minimal subset which all devices must handle greatly simplifies the hardware and provides greater compatibility between devices (especially third party devices). This is how we guarantee compatibility between devices.

Rectangular Regions

Most source devices contain Source Resource Channels which are capable of generating a stream from a rectangular region. The stream is generated by scanning the specified rectangular region from left to right and top to bottom and sending the pixels to the destination. The SRC needs no information about the destination (other than its location). It simple needs to create the stream. The source may use its own "Row Bytes", storage format, color space, etc... as long as it can generate a stream in one of the 4 legal formats.

Most destinations include Destination Resource Channels capable of translating streams into rectangular regions. As the stream is received the DRC (in the case of a frame buffer) writes the stream into memory. The DRC determines where to write the incoming stream from its registers which were set up by the CPU. The destination performs any color space translation, row bytes translation, offset, or other processing to the stream in order to correctly display it (in the case of a frame buffer). For example if a frame buffer is displaying grayscale and it receives a `k32_RGB` it is responsible for translating the data into grayscale format.

Arbitrary Shapes and Clip Alpha

So far we have only described capabilities which allow for transfers and compositing on a rectangular region basis. The Wilson Architecture provides support for arbitrary shapes via the alpha channel. All streams which are transferred include a full alpha channel. Each source must include at least a 1 bit mask equal in size to the largest rectangular region it can source. This is needed to supply information in the alpha channel. The source may contain a full 8 bit or 16 bit alpha channel if it

desires. The mask data or the alpha data must be placed into the alpha channel of the stream when a stream is sent to a DRC. The mask or alpha data is maintained by the CPU(s).

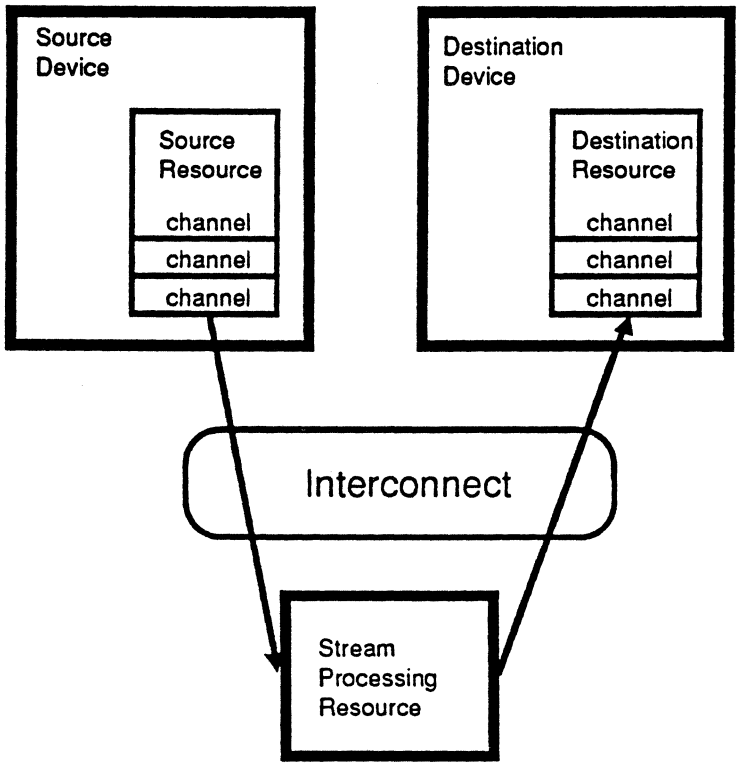
All destinations must be capable of clipping pixels (not writing them) based on the alpha channel data. If the value in the alpha channel equals zero the DRC will not write the pixel. This relatively simple functionality provides for both overlays and arbitrary shapes. The destination's mode will normally be programmable (clip or no clip).

When the alpha channel is used for clipping information as described above it is termed "Clip Alpha". When Clip Alpha is received by a destination resource the corresponding pixels will not be used if alpha equals zero.

Does not perform anti-aliasing between Regions (windows) from different sources. Can be done (optionally) in software.

Stream Processing Resources

Stream Processing Resources can be included in the system. These are simple devices which act as both a destination and a source of streams. They receive a stream, perform an operation on the stream, and send the new modified stream on to a destination. An example is shown below. Stream Processing Resources can deliver data to other Stream Processing Resources to create a chain of processing.



An example of Stream Processing Resource

It is possible for a Stream Processing Resource (SPR) to receive multiple streams and generate a single stream as its output. A simple case of a Stream Processing Resource might be a Blending Resource. The Blending Resource would receive two streams, perform an alpha based blend operation, and send the result to the destination. Note that Stream Processing Resources are relatively simple and contain little or no buffering.

Just as Source Resources and Destination Resources have multiple channels Stream Processing Resources can have multiple channels (SPRCs). These allow simultaneous processing on multiple streams.

Stream Processing Resources can be relatively independent of the final source and destination of the stream. They provide a general resource which nearly any stream in the system can be sent through.

Data Streams

Behind the basic ideas of Wilson Streams is a focus on moving images. The data types that are specifically defined are pixels. The Wilson Architecture, however, also supports the movement of plain byte streams. The byte streams could be used for the movement of sound information, the movement of compressed video, the movement of YUV pixels, the movement of ascii data, etc... The Wilson Architecture, however, does not define the format of these alternate data types. They are all simple moved as plain byte streams. Definitions may be put in place for these data types if a future need arises.

Jag1 Wilson Feature/Functionality Summary

The following functionality is planned for the Jag1 implementation of Wilson: The motherboard can act as both a source and a destination of rectangular pixel regions. It will contain a Source Resource and a Destination Resource. These can be used to access both main memory and on the on board frame buffer. They are termed the Motherboard RectRegion Source Resource (MRSR) and the Motherboard RectRegion Destination Resource (MRDR). The MRSR can send regions to other frame buffers or processors. The MRDR can receive regions from video input devices or processors. The MRSR can send a region to the MRDR to accomplish a bit blit or a rectangular region clear.

All Wilson related functionality will be implemented in the Expansion Interface and Wilson Chip (E&WC).

Motherboard RectRegion Destination Resource

- The motherboard frame buffer and system memory can act as a destination for incoming rectangular regions which are in any of the 4 Wilson stream formats.

- The motherboard can convert incoming pixels to the appropriate form. If data is received by the motherboard in any of the 4 formats it can be converted to the motherboard's frame buffer depth (Y=9/32*R + 19/32*G + 4/32*B will be used for conversion from color to gray scale and value replication will be used for conversion from gray to color). The motherboard's framebuffer may be either 24 bits RGB or 8 bits Gray.

- Pixels can be masked by their alpha (they are not written if alpha=0). This provides for arbitrarily shaped windows and overlays. This can be programmed on and off for each channel.

- Pixel data can be written in 3 different forms: 1) only store intensity data, 2) only store alpha data, or 3) store both intensity and alpha data. These are used for frame buffers which lack alpha storage, for stripping an alpha channel, or for normal operations respectively. Note that pixel twisting is performed.

- The motherboard will provide the ability to scatter data on page size address boundaries (this is needed to support virtual memory). This is accomplished using the Sequencer. See the Sequencer hardware section for details.

Motherboard RectRegion Source Resource

- The motherboard can source rectangular regions of `k16_GRAY`, `k32_RGB`, `k32_GRAY`, and `k64_RGB` when stored in memory as `k16_GRAY`, `k32_RGB`, `k32_GRAY`, and `k64_RGB` respectively (these are Single Stream Transfers).

  VALID SINGLE STREAMS (combined alpha & intensity data): `k16_GRAY`, `k32_RGB`, `k32_GRAY`, `k64_RGB`, 64 bit constant.

- The motherboard can source regions in the `k16_GRAY` and `32_RGB` formats from two separate streams (these are Dual Stream Transfers). The Dual Stream Transfers utilize intensity from one stream and alpha from a second stream to create the final stream. A Stream Processing Resource (the Pixel Munger) is used to accomplish this.

  Valid ALPHA: 1 bit, 8 bits, 8 bit constant, 8 bits from `k16_GRAY`, 8 bits from `k32_RGB`.

  Valid INTENSITY: 8 bits, 24 bits, 8 bits constant, 24 bits constant, 8 bit from `k16_GRAY`, 24 bits from `k32_RGB`.

- Constant values can be supplied for region clears & fills for both alpha & intensity.

- The motherboard will support rate control to facilitate beam chasing and bandwidth allocation.

- The motherboard will include an 8 bits per pixel to 24 bits per pixel color translator (CLUT). This can be used for expanding an 8 BPP image to a 24 BPP image during transfer. The Pixel translator will be placed in the Pixel Munger Stream Processing Resource.

- The motherboard will provide the ability to gather data on page size address boundaries (this is needed to support virtual memory). This is accomplished using the Sequencer. See the Sequencer hardware section for details.

General Wilson Functionality

- The motherboard will provide hardware for blending two streams of either `k16_GRAY` or `k32_RGB` to create a single output stream. The blend of the two channels is a multiply based on the alpha channel of the first stream. The streams must be the same type.

- The motherboard will support an engine (the Sequencer) which can walk a sequence of commands in main memory. The command lists will be used to support scatter/gather operations (page fragmentation makes this necessary) and transfers of rectilinear regions. The engine will be capable of interrupting the processor at various points during the transfer. The engine will support queueing of Wilson commands.

Expansion I/O Interface Features

The Expansion Interface and Wilson Chip will perform bus conversion between the Expansion I/O and the local XJS Bus. The following features will be implemented.

- The E&WC will perform protocol conversion for normal memory mapped transactions between the CPU(s) and the Expansion I/O.

- The E&WC will perform locked "Exchange" instructions over the Expansion I/O.

- The E&WC will perform locked "Exchange" instructions in the local memory space when requested by Expansion I/O Masters.

- All Read transactions performed by the E&WC are "split response"

- The E&WC will participate in cache coherency protocols implemented by the XJS.

- The E&WC will implement an error handling scheme that avoids dead locks and provides information for recovery software to utilize.

- The E&WC will utilize the stream protocol of BLT for stream transfers via BLT.

Implementation Summary

In order to perform the fundamental functions listed above the E&WC contains five fundamental blocks

1) Motherboard RectRegion Source Resource (MRSR). The MRSR can read a rectangular region of bytes from memory and create a stream from the data. The data stream can be sent to any Destination Resource. Destination Resources can be located on the motherboard or on the Expansion I/O. The MRSR can also source a constant valued stream.

2) Motherboard RectRegion Destination Resource (MRDR. The MRDR receives byte streams from other Source Resources. It places the data into a rectangular region in memory (or framebuffer). The MRDR can perform translation from RGB intensity to grayscale intensity and grayscale intensity to RGB intensity. The MRDR can also perform clipping operations based on the alpha values associated with the intensity. The pixels are optionally not written if alpha=0. The MRDR can be programmed to write intensity data & alpha data, only intensity data, or only alpha data. The unwanted data is eliminated from the byte stream (packing and twisting are performed).

3) Pixel Munger. The Pixel Munger is a Stream Processing Resource. It receives two streams from Source Resources. The two streams are combined to create a single output stream. The Pixel Munger accepts many different stream types corresponding to data formats which might be used by the CPU. These include: `k16_GRAY`, `k32_RGB`, 8 bit alpha only, 8 bit gray only, 24 bit RGB only, and 8 bit pseudo color. The Pixel Munger accepts two streams and creates a single output stream. The output is created by taking the alpha from one input stream and the intensity from the other input stream. The output of the Pixel Munger is always either `k16_GRAY` or `k32_RGB`. The Pixel Munger will also include an 8 bit to 24 bit color translator (CLUT).

4) Blender. The Blender is a Stream Processing Resource. It accepts two input streams and performs an alpha blend. Both streams must be either `k16_GRAY` or `k32_RGB`. The second streams pixel values are multiplied by 1-alpha of the first stream and added to the pixel values of the first stream. The second streams alpha is normally treated like the other pixel information. The Blender has a special alpha override mode. When this mode is enabled and the second streams alpha value=0 the output alpha's value is forced to 0. This is used for maintaining clip alpha information while compositing.

5) Sequencer. The sequencer does not accept or generate streams. It is not a Resource. It executes a simple instruction sequence in main memory to start and stop the transfer of streams. The instruction sequence is used to perform scatter and gather operations of data in the virtual memory environment.

User Scenario

The motherboard's Wilson Hardware will be designed to support the 4 simultaneous operations listed below on a single 16" monitor at 24 bits per pixel:

1) 1/2 screen decompressed video display with antialiased text over the live video.

2) 1/4 screen back buffered animation running at 15 Hz.

3) 1/4 screen scrolling operation from 8 BPP backbuffer.

4) Cursor carrying icon moving across screen (not antialiased).

Channels Needed For User Scenario

The hardware channels needed to support the operations listed in the user scenario are listed below:

1) needs 4 Sequence Channels, 4 MRSR Channels, 2 MRDR Channels, 1 Blender Channel, 1 Pixel Munger Channel

2) needs 2 Sequence Channels, 2 MRSR Channels, 1 MRDR Channel, 1 Pixel Munger Channel

3) needs 1 Sequence Channel, 2 MRSR Channels, 1 MRDR Channel, 1 Pixel Munger Channel

4) needs 1 Sequence Channel, 1 MRSR Channel, 1 MRDR Channel

TOTAL: 8 Sequence Channels, 9 MRSR Channels, 5 MRDR Channels, 3 Pixel Munger Channels, 1 Blender Channel.

Note that more Pixel Munger Channels may be needed if data formats are incorrect or need to be changed.

Wilson Channel Quantities

- The E&WC will implement 8 MRDR Channels
- The E&WC will implement 8 MRSR Channels
- The E&WC will implement 8 Sequencer Channels for support of scatter/gather operations.
- The E&WC will implement 4 Pixel Munger Channels for handling dual stream conversions
- The E&WC will implement 2 Blender Channels for performing alpha blending.

Alpha Compositing Overview

This section attempts to analyze how compositing of multiple images using a full alpha channel can be performed. It details some limitations when using the Blender to perform the multiplies.

Assume 4 images which need to be composited, I1, I2, I3, I4. I4 is the rear most image in the set. Each pixel of each image has an alpha value associated with it (A1, A2, A3, and A4) and a set of RGB values (RGB1, RGB2, RGB3, RGB4). The rear most image (I4) is the background and has constant alpha value of 1 (represented by 255). All images are assumed to be premultiplied (the pixel values never exceed alpha). RGBW and AW are temporary variables which are never actually placed into memory (they are streamed directly from the source to the destination). The compositing equations are shown below:

Front to back compositing:

```
RGBW1 = RGB1 + (1-A1) * RGB2
AW1   = A1 + (1-A1) *A2
RGBW2 = RGBW1 + (1-AW1) * RGB3
AW2   = AW1 + (1-AW1) * A3
RGB   = RGBW2 + (1-AW2) * RGB4
A     = AW2 + (1-AW2) * A4
```

Back to Front Compositing

```
RGBW1 = RGB3 + (1-A3) * RGB4
AW1   = A3 + (1-A3) * A4     =  1  (because A4=255)
RGBW2 = RGB2 + (1-A2) * RGBW1
AW2   = A2 + (1-A2) * AW1    =  1  (because AW1=255)
RGB   = RGB1 + (1-A1) * RGBW2
A     = A1 + (1-A1) * AW2    =  1  (because AW2=255)
```

The Blender performs the calculations on RGB and A. For the Back to Front case I4 and I3 are streamed into the Blender and the Blender generates the W1 result. The W1 result is streamed back into the Blender along with I2. The Blender generates the W2 result. The W2 result and I1 are streamed into the Blender and the final result is generated. The final result is streamed to the destination.

Clip information may be provided with the alpha data. If alpha includes clip information it is labeled as Clip Alpha. If a Clip Alpha = 0 the data should not be written to memory (it is clipped). If the Clip Alpha is not zero it is treated as normal alpha. Most external sources (video input device for example) provide clip information on the alpha channel. The clip information has been expanded to 8 bits (0 = no write, 255 = write). When compositing live video this information must be preserved in order to avoid overwriting data on the screen. When performing front to back compositing the video must be that last thing composited (e.g.. it is located at the back & has no transparent elements). A special mode is used in the blender to provide an alpha override. Alpha compositing is performed as normal except that if the video's alpha value is zero the result alpha is forced to zero. For example, if I4 was video the final composite step performed, we obtain:

```
A = AW2 + (1-AW2) * A4
```

If A4 = 0 and alpha override is enabled the value of A will be zero. If alpha !=0 compositing is performed normally. Additional compositing cannot be performed after the video with the Clip Alpha has been composited because some alphas may have been forced to zero. This basically implies that when doing front to back compositing one cannot antialias video edges with information located behind the video and one cannot have any transparent video. After the composite is completed all data which has an alpha!=0 is written to memory.

When compositing back to front the Clip Alpha concept can also be used. If the video is the rear most image it will have an associated alpha of either 0 or 255. Alpha = 255 where the pixels should be written and = 0 where they should not be written. If the video is composited with images in front of it, with the alpha override enabled for all composite steps, the proper results are obtained. For example if I4 is video we obtain:

```
RGBW1 = RGB3 + (1-A3) * RGB4
AW1 = A3 + (1-A3) * A4    =  1 or 0  (because A4=255 or 0)
```

If alpha override is enabled RGBW1 is calculated properly and AW1 is either 1 or 0 depending on the value of A4. Additional composite steps can be performed on the result as long as the alpha override is still enabled. After the result is complete the pixels with an alpha of zero (those clipped at the start) are not written to memory. In this case partially transparent video could be implemented if the clip information was provided with the first image composited (the background). Further compositing can be performed on top of the background provided each image contains a full alpha channel and the clip information will not be lost.

Section 3

Wilson High Level Software

This section discusses some higher level software issues which will affect the lower level software and the hardware. I am really only concerned with the affect on hardware. I may outline solutions for some of the problems I see, but these solutions are outlined primarily to convince myself that the problem can be solved and that the hardware being provided is useful. The solutions are not meant to be all encompassing.

Real Time

It is meaningless to discuss the use of Wilson without first understanding the requirements of real time and at least superficially how they are met. "Real time" has historically meant responding to an interrupt or other external event within a fixed amount of time. The response time has been labeled as the "real time response." This concept, although extremely useful when software is performing direct hardware control (turning motors on and off), is not the concept of real time which is needed in Jaguar.

The Real Time (now in capitals) which we are concerned with is very different than a simple real time response. Certain operations performed by computers have a relationship with reality. Video, animation, sound, modem operation, voice reproduction, and scrolling are examples which must progress at a fixed speed regardless of the CPU's speed. When these media types are presented to the user at the proper speed the system is said to be operating in Real Time. If animation is synchronized to the sound it must not progress too slowly or too quickly. It has a correspondence to reality which must be maintained (the bird should move across the screen in 8.46 seconds). It is not acceptable for the animation to slow down if CPU cycles are not available. It is not acceptable for the live video to slow down if the bandwidth is not available. It is not acceptable for the sound to slow down or skip. It is, however, acceptable for the quality of any of the above to degrade in a less objectionable fashion which maintains Real Time synchronization. If the CPU or another system resource is overloaded the quality can be reduced in order to reduce the load. The quality can be reduced by dropping frames in video, by simplifying the rendering models in animation, or by reducing the processing (mixing, filtering, or sample rate conversion) in sound.

DISCLAIMER: Animation does not always have to be Real Time. Relative to the issues here, however, Non-Real Time Animation is not interesting. It is not significantly different than a standard application. When we refer to animation in the remainder of this document Real Time animation is implied.

Graceful Degradation

Graceful degradation is the reduction of resource utilization in a non-catastrophic manner. Graceful degradation can be observed in many different areas. It is crucial to a Real Time system because it is the only way to handle an overload and maintain a Real Time environment.

In a Macintosh environment when the CPU becomes overloaded it simply performs tasks more slowly. This is a graceful form of degradation which a user can understand. It would not be acceptable, for example, for an application to quit and give a message such "Application XXXX unexpectedly ended" if it couldn't obtain the CPU resources it really needed. Doing so would be stupid. In a system attempting to perform Real Time operations graceful degradation is more complex. It is still acceptable for non-realtime applications to degrade by slowing down. Real Time applications, however, must gracefully degrade in other manners. These methods are not nearly as automatic as the slow down caused by lack of CPU cycles.

If graceful degradation methods are not used in a Real Time application problems which are not intuitive may present themselves. These include losing synchronization between video & sound, gaps or clicks in the sound, display of partial images, etc... Graceful degradation methods must be utilized for a robust Real Time system. A large number of methods exist for Real Time applications to gracefully degrade. These include using simpler rendering algorithms, performing less exact calculations, using less accurate models, using fewer sound channels, performing simpler sample rate conversion, dropping frames, etc... The list is very long. These methods reduce resource utilization without losing the connection and synchronization to Real Time. These methods are also somewhat intuitive to the user; they are graceful.

Jaguar will attempt to provide some of the basic mechanisms needed to obtain graceful degradation of Real Time processes. The difficult part, however, is creating a software environment that can handle graceful degradation.

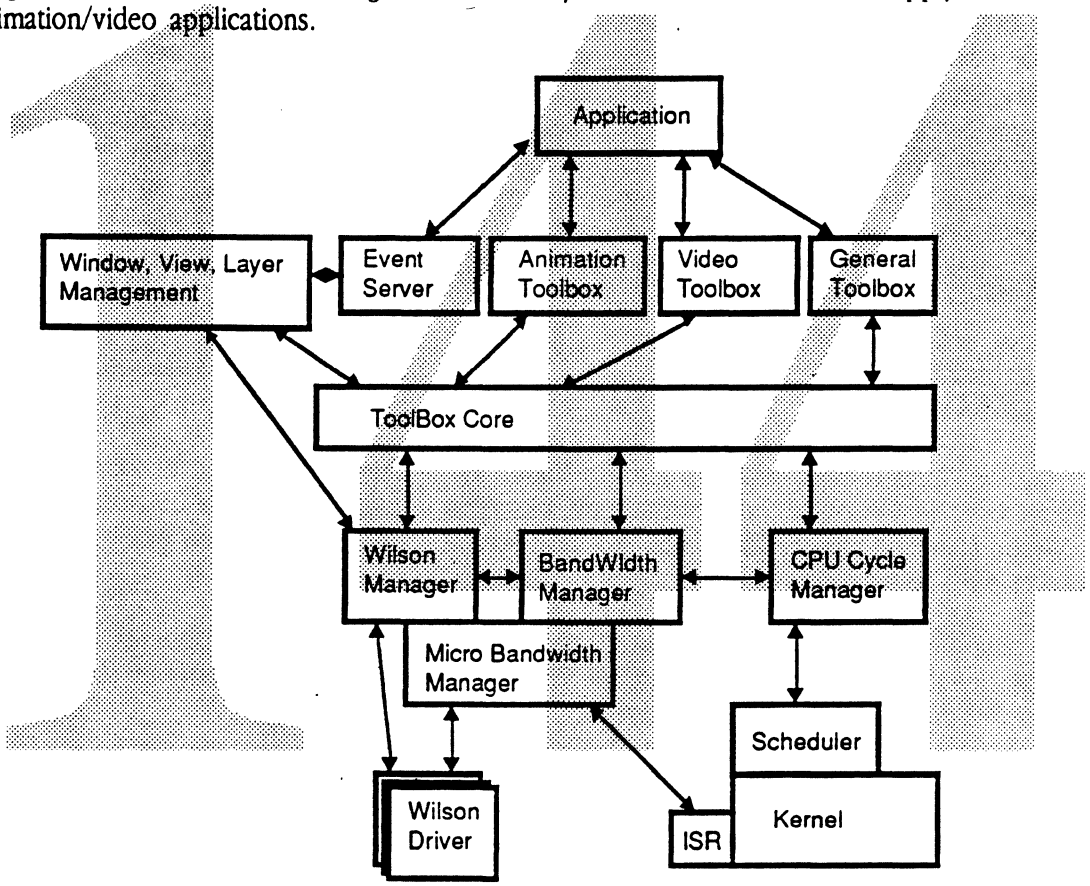
Allocation of CPU Cycles and BandWidth

Bandwidth and CPU cycles must be allocated prior to execution if Jaguar is to maintain a "Real Time" environment. Bandwidth should be allocated and controlled for each device individually. Each device must have a parameter which describes its maximum bandwidth capability. For example a bus might have a parameter of 400Mb/sec total transfer rate or a Frame Buffer might have a parameter of 100Mb/sec total input and output. These parameters are only estimates. Bandwidth and CPU Cycles cannot be allocated down to the last the cycle. This is unrealistic given the number of interactions in the system. The bandwidth allocation and CPU Cycle allocation, however, will hopefully place us within 10%. If we can guarantee 10% accuracy and we leave a 10% margin everything will work as predicted. This is not meant to imply we do not need recovery mechanisms. Recovery mechanisms will still be necessary for when software or applications fail to properly handle allocation.

The alternative to approximate allocation is no system resource allocation. The Recovery Mechanism are simply invoked to handle things when they break. This is not acceptable because it does not provide for graceful degradation of the system while maintaining a Real Time environment.

A View Of Software

In order to discuss how the hardware is used to solve some of the problems with performing Real Time operations a software model is needed. Below is a naive diagram of a possible software system. The diagram's only purpose is to provide a discussion mechanism. How the final implementation will be partitioned is unknown at this time. I do not know exactly how this diagram might map into the Pink environment. Each of the blocks and their interactions are discussed in the following sections. The interactions diagrammed are only those which are known to apply to Wilson and animation/video applications.



A Simple View of a Software System

Animation Toolbox

The Animation Toolbox is part of the System Software Toolbox. It interacts with applications which perform both animation and Real Time animation. It provides services to make an application writer's job easier. Exactly what services the Animation Toolbox could/would provide will not be described here. The Animation Toolbox, however, is an animation application's interface to system software. It communicates with the BandWidth Manager, the Wilson Manager, and the CPU Cycle Manager via the Toolbox Core to obtain system resources. The relationship with these

managers is two way. The Animation Toolbox will request resources which may be granted. These resources may allow a high quality animation to be performed by the toolbox. If system resource allocations must be changed during the animation the Animation Toolbox is warned by the CPU Cycle Manager, the Bandwidth Manager, or the Wilson Manager of the resource removal. The Toolbox (or application) must acknowledge the change and act accordingly. The Animation Toolbox may choose to operate in a degraded mode (using less resources) or pass the information up to the application.

When making requests for resources the Animation Toolbox needs to know what type of resources are needed. It may need information from the layer manager, the window system, the graphics toolbox, and the application to determine the needed resources. Once it knows the needed resources it can make requests of the Managers via the ToolBox Core. The requests to the managers include back off points which may be granted by the managers if all the resources desired by the toolbox are not available. If a back off point is granted to the toolbox the application may have to be informed. If the user moves windows on his screen the requests may be modified. The ToolBox Core may have to cancel requests sent to the Managers based on information received from the Window/View/Layer system and make new requests. These new requests may or may not be granted. The application may have to be informed if resources are no longer available.

Video Toolbox

The Video Toolbox provides services similar to the animation toolbox for video. It makes requests for system resources and maintains the push/pull relationship with the CPU Cycle Manager, the Wilson Manager, and the Bandwidth Manager via the ToolBox Core. It too will be informed if resources must be removed.

The Video Toolbox is responsible for graceful degradation as resources become less available. It should transparently degrade the video quality if resources are removed (by one of the managers) or the resources can not be allocated.

ToolBox Core

The ToolBox Core is responsible for a lot of things. In this environment it is primarily responsible for combining a toolbox's request and information from the Window/Layer/View System to generate a request to Wilson. The clipping information from the Window/Layer/View System will potentially (most likely) impact the request. Clipping will often times greatly reduce the resources required.

The ToolBox Core is also responsible for changing transfer requests when windows or such are moved around (clipping information changes). For example if a window is popped to the front the clip information will change. This might require a larger transfer to be performed from the video input to the frame buffer. The ToolBox Core must modify its request to the Wilson Manager (eliminate the old request and make a new request). The ToolBox Core may have to inform the original requesting toolbox (and eventually the application) if resources are not available.

Window, Layer, and View Manager

The Window, Layer, View Manager controls who owns what pixels on the screen. It receives events to change this. If the screen allocation changes it must inform the ToolBox Core such that it can change the Wilson Transfer's which are occurring.

Event Server

The Event Server delivers events to the Window, Layer, and View Manager and to the Application & toolboxes.

CPU Cycle Manager

Applications or tasks which have real time requirements register their needs with the CPU Cycle Manager via the appropriate toolbox. The CPU Cycle Manager interacts with the Scheduler to determine what tasks are executed.

When an application or Toolbox requests resources they are granted if they are available. If the resources cannot be granted a push/pull type of operation begins between the CPU Cycle Manager and the Toolbox to resolve the problem. When a request is made of the CPU Cycle manager information about back off points is provided such that the Cycle Manager knows how much of the resources can be removed from the requester at a later time. This may be necessary if the priority of operations in the system changes. Information on how back off should be handled is also provided to the CPU Cycle Manager (how the application should be informed that resources are going to be removed).

When resources which have already been granted must be removed the CPU Cycle Manager must inform the application (or toolbox) before they are removed. Exactly how this will be handled must be determined.

Scheduler

The Scheduler controls which tasks or applications obtain which portion of the CPU's cycles. It uses information from the CPU Cycle Manager to determine which task should execute and for how long. It is generally considered part of the Kernel. In the case of OPUS it may sit on top of the Kernel.

Applications which do not have real time constraints (have not requested CPU Cycle Allocations) are most likely placed at the bottom of the priority queue for execution. Some exceptions to this, however, will most likely exist.

Bandwidth Manager (BWM)

Bandwidth allocation is performed by the Bandwidth Manager. Bandwidth is managed in a fashion similar to CPU cycle management.

Bandwidth allocation works except when the CPU decides to perform some memory transactions. The solution to this fundamental problem relies on absolute unfair priority. The CPU always operates at the lowest possible priority. All bandwidth allocated transfers operate at a higher priority and can completely block the lower priorities. This allows the CPU to use all excess bandwidth, but allows bandwidth allocation to function properly.

The Bandwidth Manager performs operations similar to the CPU Cycle Manager. It, however, allocates Bus and other Resource Bandwidth. It's allocations are extremely hardware dependent. The Animations Toolbox and the Video Toolbox request Bandwidth Allocation from the Bandwidth Manager via the Toolbox Core. The Toolbox Core requests BandWidth either directly or through the Wilson Manager. Most likely the request is made through the Wilson Manager. The Toolbox Core describes the transfers needed to the Wilson Manager. The Wilson Manager attempts to allocate the Bandwidth needed via the Bandwidth Manager. It returns the result (possibly a compromise) to the requesting toolbox.

Each device in the system has a Maximum bandwidth parameter, a Continuous allocated bandwidth parameter, and a Instantaneous allocated bandwidth parameter. The Maximum bandwidth parameter is hardware dependent and is fixed. The Continuous bandwidth is maintained by the Bandwidth Manager. The Instantaneous Bandwidth parameter is maintained by the Micro Bandwidth Manager. Neither of the two bandwidth parameters can ever exceed the Maximum bandwidth. The maximum bandwidth parameter is probably not a true maximum. An average maximum throughput value might be used instead.

Before any data movement transactions can occur a request must be made with the Bandwidth Manager (BWM). Three classes of transactions can be requested from the BWM: 1) Continuous, 2) Repetitive Burst, and 3) OneTime. A Continuous transfer is one which has a relatively constant bandwidth and which has a fixed peak rate. It might be used for sound channels or low quality video. A Repetitive Burst transfer is one which operates in real time. It may have some Continuous characteristics but its basic mode of transfer is large high bandwidth bursts at a fixed frequency. It might be used for animation or high quality video. OneTime transfers are single operations which are allocated, performed, and deallocated. They do not usually have any constraints on their execution time.

Continuous Bandwidth Transactions: The BWM receives a request for some Continuous Bandwidth between two devices. It checks the total amount of bandwidth allocated to the two devices and any connection paths. If the Bandwidth is available at/through all the devices the BWM updates their Continuous allocated bandwidth parameters and returns. If the Continuous Bandwidth cannot be allocated because a large amount of Repetitive Burst Bandwidth is allocated the BWM may choose to wait until some of the burst bandwidth is deallocated. If the Continuous BandWidth cannot be allocated because a large amount of Continuous Bandwidth is already allocated the BWM may choose to reduce another requester's bandwidth. It will send a message to the requester informing it of the needed reduction. The requester must send a reply in a specified minimum amount of time or the transfers will be terminated completely. Continuous bandwidth is deallocated when the BWM receives a request to de-allocate it.

Repetitive Burst Transactions: The BWM receives a request for Repetitive Burst Bandwidth. The request specifies the source, the destination, the rate of transfer, the length of the transfer, and the frequency of the transfer. The BWM determines if the request is reasonable and how well it can be met. If the request can be met it is granted. If not the push/pull type of bargaining is performed between the requester and the BWM.

OneTime Transfers: The BWM receives a request for a transfer of data. It is a one time transfer. If the request has no constraints (speed of transfer or such) it is granted at a specific rate. This is probably the format of a normal bit blit operation (it can be done at any speed). Different reply methods can be utilized for the blit (basically synchronous or asynchronous).

Wilson Manager

The Wilson Manager receives requests from the toolboxes for data transfers. These requests include constraints about when the transfers can be performed, acceptable back off methods, reply methods, and synchronizations methods. The back off methods describe what should be done if the constraints cannot be met. The reply methods describe what messages should be sent when the operation is complete, if the operation cannot be performed, or after each burst has been completed. The synchronization methods describe how the Wilson Manager should synchronize with other objects or tasks in the system.

The Wilson Manager knows what type of hardware the system has available (what Wilson Driver's are available). The hardware may provide additional resources which the Wilson Manager can use to meet the toolbox's request. This will affect the bandwidth required. The hardware available will affect the method the Manager uses as well as the operations the Wilson Manager can perform (its very hard to perform live video without video input!). If the hardware or channels are not available to perform the operation requested and the operation cannot be emulated adequately in software the Wilson Manager informs the toolbox via the reply methods.

Once the hardware and resources have been located the Wilson Manager attempts to obtain the bandwidth needed from the BWM. If the bandwidth cannot be allocated the Wilson Manager uses the back off methods to obtain a compromise. The compromise information is returned to the requester using the reply methods. If a compromise cannot be obtained the Wilson Manager informs the requester.

Once the Wilson Manager has obtained the bandwidth needed it begins to synchronize the transfers and perform those whose constraints have been met. It works the with the Micro Bandwidth Manager when performing the transfers.

The Wilson Manager is responsible for translation from virtual address space to physical address space. The translations are performed by the Wilson Manager and relayed to the MBWM and the BWM when making requests. The BWM and MBWM must know what physical piece of hardware the transfer is being made to. The Wilson Manager must use physical addresses when communicating with the Wilson Drivers.

Rectangular regions received by the Wilson Manager must already be clipped by the window/layer manager. The Window manager must update clip masks on the source. It must know the source of the Window's contents!

CPU Cycles & Their Relationship To BandWidth

The bandwidth required by the CPU is not predictable. This introduces a fundamental problem when attempting to allocate bandwidth in the system. Priority is used to partially solve this problem, but their is still a fundamental relationship between CPU cycles and Bandwidth. If a transfer is occurring which requires a large percentage of the available bandwidth to system memory which has been allocated (operating at a high priority) the CPU's performance may degrade. How does the CPU Cycle Manager account for this degradation? The degradation, unfortunately, is extremely code dependent.

The only known solution to this problem is to use a heuristic algorithm for CPU performance degradation based on bandwidth allocation.

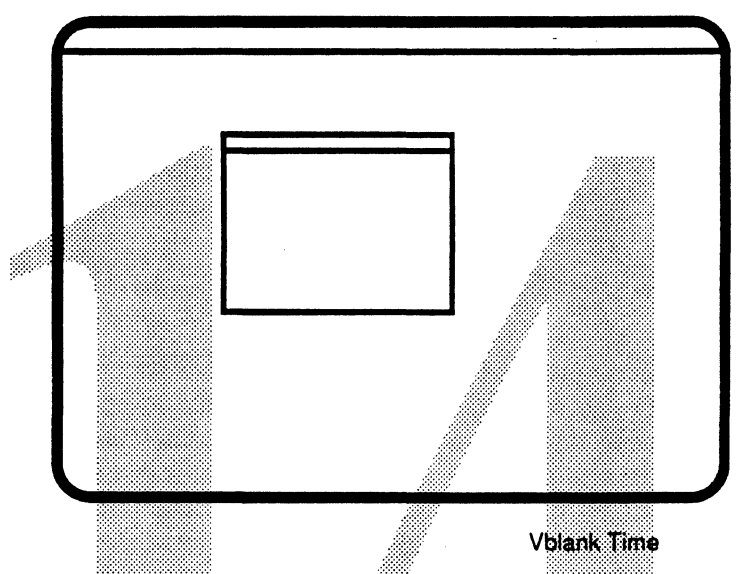
Priority is used to reclaim some of the Bandwidth which must be allocated but may not be used. All transactions which are allocated are given a priority which is greater than 0. Transactions which operate at a priority of zero do not therefore have to be allocated. This has some important implications on the implementation of priority in the system. The CPU generally operates at priority 0. It is able to reclaim almost of the unallocated bandwidth. Simple blit operations could also be executed at a priority of 0 if their execution speed was not important.

Tear Free Updates

Tear free updates of animation and video are desired in Jaguar. If video is being displayed in a region on a monitor it is possible to generate a frame tear. A frame tear occurs when the monitor displays in a single frame two different video frames. If video is placed in the frame buffer without regard to the location of the monitor's beam tears will be generated. Tear's can be avoided by performing what is sometimes called "beam chasing." Frame tears can also occur when performing animation. In fact frame tears can even affect such simple things as the cursor. The Macintosh avoids tearing the cursor by only changing the cursor during the vertical blanking interval. This results in a high quality solid cursor which does not flicker. Vertical blank is long enough to update an object about the size of a cursor. If the cursor picks up a window and drags it around frame tears can be observed (vertical blank is not used for the window outline).

A general solution for avoiding frame tears in animation and video is desired. A solution will be outlined here. Below is a diagram of a monitor screen with one window displayed. We assume the monitor's raster is traced from left to right and top to bottom (as one would read a page of text). This is true of all monitors made by Apple today (and everyone else I know of). At any moment in time the raster will have a Y coordinate. If we assume the contents of the window can be transferred to the frame buffer in a fixed maximum amount of time we can determine the "Safe Region" for beginning the transfer. If the raster is in the upper "Safe Region" and we begin transferring to the window (we assume that the transfer to the window is also top to bottom and right to left) the raster

beam will not catch up to the transfer. No tears will be generated. If the raster is in the lower "Safe Region" the transfer to the window will never catch up to the raster and no tears will be generated. If the raster is not in the "Safe Region" the raster will cross the transfer and two different frames will be shown in a single monitor frame -- A tear is created.



Monitor Screen with Window Showing Safe Region to begin an update

In the above example a number of assumptions are made: 1) The transfer rate of data is high enough such that avoiding tears is possible and 2) A fixed transfer rate is selected in order to determine the "Safe Region". If a fixed transfer rate is not assumed the "Safe Region" will vary (their is actually a mathematical relationship). The "Safe Region" is dependent on the window's geometry, the window's location, and the monitors retrace rate.

When placing live video onto a monitor screen the problem is further compounded by the input video beam's trace. The input video effectively has a raster beam which is tracing the input. It too will have a "Safe Region" determined by the location of its raster, the speed of the transfer, and the geometry. Both rasters (frame buffer and video input) must be in their safe regions when the transfer is started to avoid a frame tear.

The hardware planned for all frame buffers and all video input devices will contain several programmable interrupts (8 for Jag1 motherboard) which can be programmed to occur at any vertical raster position. These interrupts can be programmed to occur when the "Safe Region" is entered. The interrupts are used by the Wilson Manager and the Micro BandWidth Manager to avoid frame tears when transferring regions of data.

Micro Bandwidth Manager (MBWM)

The Micro Bandwidth Manager allocates bandwidth in extremely small sections of time. It is very tightly coupled to the Bandwidth Manager, the Wilson Manager, and the Wilson Drivers.

The BWM allocates bandwidth over long periods of time. It receives requests for Continuous Bandwidth or for Repetitive Burst Bandwidth. The MBWM manages bandwidth at a much lower level. It manages bandwidth at the single burst level rather than the repetitive burst level. The BWM gives the go ahead to the toolbox or application that the request it has made is reasonable and that the system should be able to meet the needs. The MBWM schedules the transfer such the constraints given to the Wilson Manager are met. The Micro BandWidth Manager works with the Wilson Manager to obtain tear free updates. An example will clarify the operation of the three components:

EXAMPLE: An application desires to place a live video window onto a screen region. It makes a request to the video toolbox. It requests Video data to be transferred from the video card to the screen at a 30 frames/sec without frame tears. It specifies the size of the source and the destination. It indicates that it wishes to synchronize to the video on a frame by frame basis. It indicates that a potential back off point is 15 frames/sec. The Video Toolbox interprets the request and makes a request of the Wilson Manager. The Wilson Manager determines the resources and bandwidth needed for the transfer. It makes a request of the BWM for the bandwidth allocation. If the bandwidth is not available it haggles with Bandwidth Manager to obtain a compromise. The Bandwidth manager may have to send messages to other tasks operating in the system informing them to begin operation in a degraded mode (it removes some of their resources). The Wilson Manager obtains the bandwidth from the BWM. It obtains Wilson Resources (channels) from itself or the Wilson Drivers. Once it has obtained a set of system resources to perform either the toolbox's request or a degraded mode of that request it returns the information to the toolbox. The toolbox may or may not pass information to the application.

The Wilson Manager sets up the channels in the proper Wilson Drivers for the transfer. It sets up any interrupts it needs to determine when the "Safe Region" has been entered. The MBWM and the Wilson Manager now interact to perform the transfer of data requested. When all the constraints for the transfer have been met (rasters are in the "Safe Regions") and the MBWM says the bandwidth is available the Wilson transfer is started. The MBWM marks the Instantaneous bandwidth as allocated. The Wilson Manager and the MBWM use the Wilson Drivers to start the transfer. When the transfer is complete the bandwidth is deallocated (an interrupt is generated at the end of the transfer for this purpose). The Wilson Manager sends a message back to the application/toolbox indicating that the frame has been transferred. The Wilson Manager and MBWM will not transfer another frame until a message is received from the application/toolbox because of the constraints. The Wilson Manager and MBWM will not transfer another frame until all the constraints are met. One of the constraints is the receipt of a message from the application.

Note that the MBWM allocates and deallocates bandwidth for each burst transfer where the BWM allocates overall bandwidth. The MBWM manager uses information about Continuous bandwidth allocation from the BWM as its starting point. The MBWM must also be informed when ever the BWM makes any allocation changes. The MBWM and the Wilson Manager can have many requests outstanding which are all waiting for constraints to be met. It may be impossible to meet some of the constraints (a deadline for example). If this occurs the MBWM or the Wilson Manager would use the reply method which was requested by the toolbox/application to indicate failure.

Wilson Manager Software Interface

This is a first attempt at trying to describe a high level software interface for managing the transfer of streams in the system. The Wilson Manager provides two primary classes of objects:

1) Streams

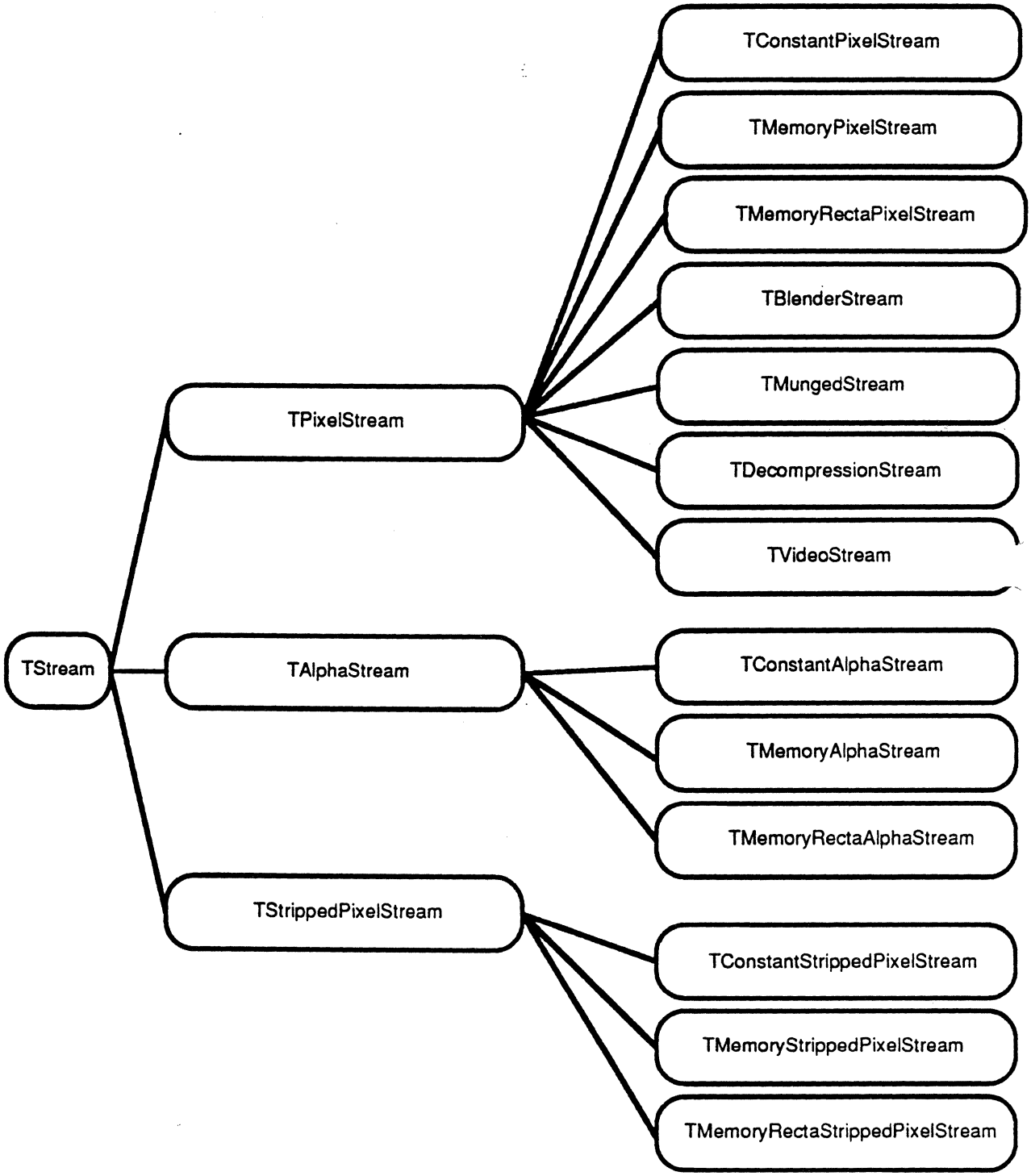
2) Stream Destinations

Streams generally correspond to the Wilson Source Resources and the Wilson Stream Processing Resources. Their "result" or "output" is always a stream. A stream is generally connected with a Stream Destination to perform an operation. A Stream Destination generally corresponds to a Wilson Destination Resource. A simple example of moving data from one location in memory to another location in memory may clarify how this works:

> EXAMPLE: To move some data from one region of memory to another region of memory a stream must be created. An object of class TMemoryRectaPixelStream is created. TMemoryRectaPixelStream is a class derived from TPixelStream which in turn is derived from TStream. TMemoryRectaPixelStream's constructor requires information about the location of the rectangular region in memory (starting address, length, width, and rowbytes for example). Once the source stream is created (in our case the object of class TMemoryRectaPixelStream) it must be routed to a Stream Destination. Our Stream Destination will be an object of class TMemoryRectaStreamDestination because we are going back into memory. TMemoryRectaStreamDestination's constructor requires an input stream (in our case the TMemoryRectaPixelStream object we created) and parameters which describe the destination (address, length, width, and rowbytes). Once the two objects are created and connected in this fashion the Stream Destination object is "activated" and the Wilson Manager transfers the data.

The Wilson Manager provides a number of classes which correspond to Source Resources. The Hierarchy of the stream objects is shown below. `TStream` is the base class from which all stream types are derived. Three fundamental classes are derived from it: 1) `TPixelStream` 2) `TAlphaStream`, and 3) `TStrippedPixelStream`. The Wilson Architecture stream types all fit in the `TPixelStream` class because they all contain both pixel intensity information and alpha information. The `TAlphaStream` contains only alpha information. The `TStrippedPixelStream` contains only pixel intensity information. The non Wilson Architecture stream classes are defined for handling data types which are appropriate to the

motherboard's main memory formats. These types are not part of the fundamental Wilson Architecture but are part of the Jag1 implementation of Wilson.



Object Derivations From TStream Base Class

All Streams are derived from the base class TStream. A derived class is used to describe how the stream is actually created, but each TStream class is capable of "sourcing" a stream of data. For example a `TMemoryAlphaStream` describes a stream in memory which contains only alpha information. The headers for the `TStream` derived classes are described below.

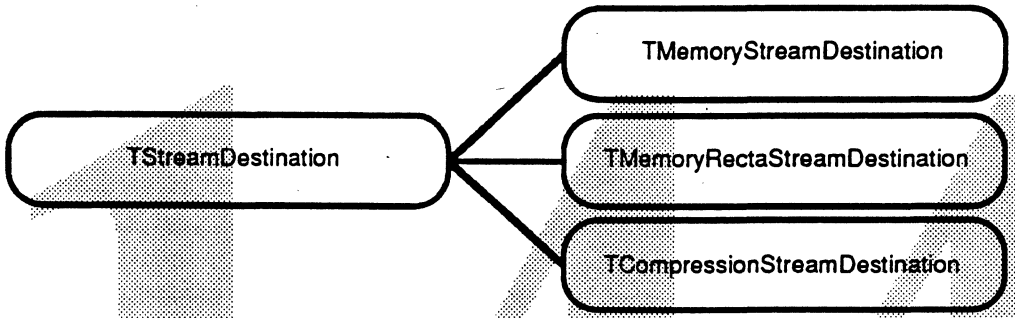
```
// Valid stream types which contain both alpha and intensity information
enum PixelStream {    k16_GRAY
                      k16_PSEUDO
                      k32_RGB
                      k32_GRAY
                      k64_RGB
                      }

// valid stream types which contain only alpha information
// these stream types are used by the Pixel Munger.  The k1_ALPHA stream type
// is for 1 bit mask information.  The k8_ALPHA stream type is an 8 bit alpha
// only stream (their is no pixel intensity information).  The k16_alpha stream
// type is a 16 bit alpha only stream.
enum AlphaStream {    k1_ALPHA
                      k8_ALPHA
                      k16_ALPHA
                      }

// valid stream types which contain only intensity information.  These are the
// pixel intensity only streams.  These streams do not contain any alpha
// information are used primarily by the pixel munger.  The k24_RGB and k48_RGB
// stream types are twisted in memory.
enum StrippedPixelStream {    k8_GRAY_NO_ALPHA
                              k8_PSEUDO_NO_ALPHA
                              k24_RGB_NO_ALPHA
                              k16_GRAY_NO_ALPHA
                              k48_RGB_NO_ALPHA
                              }


// type definitions used in prototypes
typedef unsigned long RowPixels;
typedef unsigned long RowDifference;
typedef unsigned char BitAddress;
typedef unsigned long Width;
typedef unsigned long Length;
typedef void *Address;
typedef unsigned char Priority;
typedef unsigned char AllocationResultCode;


// container class for groups of streams.
// TStreamCollection is a base class used to make a collection of streams.  It
// will might be used by the Wilson Manager to collect all the active streams
// in the system.
class TStreamCollection : public TCollectible {
    private:
    public:
        TStreamCollection();
        virtual ~TStreamCollection();
        }
```

```
// base class for all streams -- this is an abstract class
// all streams are derived from this class
class TStream : public MCollectible {
    private:
    public:
        TStream();
        virtual ~TStream();
        }

// base class for all pixel streams which include alpha and intensity info.
// this is an abstract class.
class TPixelStream : public TStream {
    private:
    public:
        TPixelStream();
        virtual ~TPixelStream();
        virtual PixelStream Type() = 0;
            }

// base class for all pure alpha streams
// this is an abstract class
class TAlphaStream : public TStream {
    private:
    public:
        TAlphaStream();
        virtual ~TAlphaStream();
        virtual AlphaStream Type() = 0;
            }

// base class for all pixel streams which do not include alpha
// this is an abstract class
class TStrippedPixelStream : public TStream {
    private:
    public:
        TStrippedPixelStream();
        virtual ~TStrippedPixelStream();
        virtual StrippedPixelStream Type() = 0;
            }

// The TConstantPixelStream is a constant value stream of a fixed length
// which contains both alpha and intensity information
class TConstantPixelStream : public TPixelStream {
    private:
    public:
        TConstantPixelStream(TColor, Length);
        virtual ~TConstantPixelStream();
        PixelStream Type();
            }

class TConstantAlphaStream : public TAlphaStream {
    private:
    public:
        TConstantAlphaStream(Alpha,Length);
        virtual ~TConstantAlphaStream();
        AlphaStream Type();
            }

class TConstantStrippedPixelStream : public TStrippedPixelStream {
    private:
    public:
        TConstantStrippedPixelStream(TColor,Length);
        virtual ~TConstantStrippedPixelStream();
        StrippedPixelStream Type();
```

```
        }

// A TMemoryPixelStream is a stream of pixels stored linearly in memory which contains
// both alpha and intensity information.
class TMemoryPixelStream : public TPixelStream {
    private:
    public:
        TMemoryPixelStream(PixelStream, Address, Length);
        virtual ~TMemoryPixelStream();
        PixelStream Type();
            }

class TMemoryAlphaStream : public TAlphaStream {
    private:
    public:
        TMemoryAlphaStream(AlphaStream, Address, Length);
        virtual ~TMemoryAlphaStream();
        AlphaStream Type();
            }

class TMemoryStrippedPixelStream : public TStrippedPixelStream {
    private:
    public:
        TMemoryStrippedPixelStream(StrippedPixelStream, Address, Length);
        virtual ~TMemoryStrippedPixelStream();
        StrippedPixelStream Type();
            }

// A TMemoryRectaPixelStream is a pixel stream which is a sequence of rectalinear
// regions in memory.  Each region has an associated start address, length, width
// and rowbytes.
class TMemoryRectaPixelStream : public TPixelStream {
    private:
    public:
        TMemoryRectaPixelStream(PixelStream, Address, Length, Width, RowPixels);
        // other constructors based on Albert Regions....
        virtual ~TMemoryRectaPixelStream
        PixelStream Type();
            }

class TMemoryRectaAlphaStream : public TAlphaStream {
    private:
    public:
        TMemoryRectaAlphaStream(AlphaStream, Address, Length, Width, RowPixels);
        TMemoryRectaAlphaStream(Address, BitAddress, Length,
                                    Width, RowPixels, RowBits);
        // other constructors based on Albert Regions....
        virtual ~ TMemoryRectaAlphaStream();
        AlphaStream Type();
            }

class TMemoryRectaStrippedPixelStream : public TStrippedPixelStream {
    private:
    public:
        TMemoryRectaStrippedPixelStream(StrippedPixelStream,
                                    Address, Length, Width, RowPixels);
        TMemoryRectaStrippedPixelStream(Address, BitAddress, Length,
                                    Width, RowPixels, RowBits);
        // other constructors based on Albert Regions....
        virtual ~ TMemoryRectaStrippedPixelStream();
        StrippedPixelStream Type();
            }
```

```
enum BlenderMode { kNormal
                   kAlphaOverride
                   }

// A TBlendedStream a full pixel stream which contains both alpha and pixel intensity
// information.  It is derived from TPixelStream.  It is constructed from two
// TPixelStreams.  A blend operation is performed on the two input streams to
// obtain the output stream.
class TBlendedStream : public TPixelStream {
    private:
    public:
        TBlendedStream(BlenderMode, TPixelStream a, TPixelStream b);
        TBlenderStream(TPixelStream);
        virtual void ~TBlendedStream;
        PixelStream Type();
        }

// A TMungedStream is  a full pixel stream which contains both alpha and pixel
// intensity information.  It is derived from TPixelStream.  It is constructed from
// two input streams.  The first input stream can be either a TPixelStream or a
// TStrippedPixelStream.  The second input stream can be either a TPixelStream or a
// TAlphaStream.  The output stream is created using the Pixel value from the first
// stream and the alpha value from the second stream.
class TMungedStream : public TPixelStream {
    private:
    public:
        TPixelStream TMungedStream(TPixelStream a);
        TPixelStream TMungedStream(TPixelStream a, TPixelStream b);
        TPixelStream TMungedStream(TPixelStream a, TAlphaStream b);
        TPixelStream TMungedStream(TStrippedPixelStream a, TPixelStream b);
        TPixelStream TMungedStream(TStrippedPixelStream a, TAlphaStream b);
        virtual void ~TMungedStream;
        PixelStream Type();
        void SetClut(unsigned char);
            }

// Need to integrate with Albert Clut Management Objects for loading CLUT.....


// TVideoStream is used to input a live video pixel stream from a video input
// device.  Its complete definition is unknown at this time.
class TVideoStream : public TPixelStream {
    private:
    public:
           TVideoStream();
           virtual ~TVideoStream();
           // unknown additions....
           }

// TDecompressionStream is used to source a pixel stream from a decompression
// device.
class TDecompressionStream: public TStrippedPixelStream {
    private:
    public:
           TDecompressionStream();
           virtual ~TDecompressionStream();
           // unknown additions.....
           }
```

The Stream Destination object class hierarchy is shown below. The TStreamDestination Class is the base class for all TStreamDestinations. Thier are currently three derived classed envisioned for each of the three potential destinations. Stream Destinations do not include Stream Processing Resources. These are included in the TStream class hierarchy.



Derivation of Destinations from TStreamDestination Class

```
// abstract class for all possible stream destinations
class TStreamDestination {
    private:
    public:
            TStreamDestination(TSchedule);
            virtual TStreamDestination();
            // additional virtual functions....
            }

enum ClipMode {   kNone
                  kAlphaClip
                  }

enum WriteMode {  kPixel
                  kAlpha
                  kStrippedPixel
                  }

// class for a stream destination which will go directly in to virtual memory.
// the address is virtual.
class TMemoryStreamDestination : public TStreamDestination {
    private:
    public:
            TMemoryStreamDestination(TStream,ClipMode,WriteMode
                                     Address, Length);
            virtual ~TMemoryStreamDestination();
            }

// class for a rectangular stream destination.
// all addresses are virtual
class TMemoryRectaStreamDestination : public TStreamDestination {
    private:
    public:
            TMemoryRectaStreamDestination(TStream,ClipMode,WriteMode
                                    Address, Length, Width, RowPixels);
```

```
        // other constructor methods based on Albert Regions....
        virtual ~TMemoryStreamDestination();
        void AddRegion(Address, Length, Width, RowPixels);
        }


// Object class for a video compression device
class TCompressionStreamDestination : public TStreamDestination {
    private:
    public:
        TCompressionStreamDestination(unknown.....);
        virtual ~TCompressionStreamDestination();
        // unknown addtions....
        }
```

Objects of the Stream class and the Stream Destination class are connected to describe the data and the operation to be performed. The Stream Destination Object is used to construct a `TSchedule` object. The `TSchedule` object describes when the transfer should occur, what type of constraints apply to the transfer, what type of back off points can be utilized, what type of synchronizations is necessary, and what type of information should be sent back to the requester.

The `Tschedule` object encapsulates all the information about a given transfer request.

```
// abstract class which is base class for all schedule types.
class TSchedule{
       private:
       public:
       TSchedule(TStream);
       virtual ~TSchedule();
       void    SetConstraints(TConstraints);       //general constr, default=none
       void    SetReplyFormat(TReply);
       void    SetDegrade(TDegrade);
       void    Activate();                         // turns transfer on and off
       void    Deactivate();
       AllocationResultCode  Allocate();
       void    Deallocate();
               // more .....
       }


//Schedule Object with minimum constraints, a one time transfer.
class TNormalSchedule : public TSchedule {
    private:
    public:
        TNormalSchedule(Priority, Tconstraints, Treply, TDegrade);
        virtual ~TNormalSchedule();
        void    SetGlobal(Boolean);
        void    SetPriority(Priority);
        }

//Schedule object for repetitive bursts
class TRepetSchedule : public TSchedule {
    private:
    public:
        TRepetSchedule(Priority, Tconstraints, Treply, TDegrade);
        virtual ~TRepetSchedule();
        void    SetFrequency(long cyclespersec);    // frequency for repetitive bursts
        void    SetFrequency(float frequencyHz);
        void    SetGlobal(Boolean);
        void    SetPriority(Priority);
```

```
    }

//Schedule object for continuous transfers
class TContinuousSchedule : public TSchedule{
    private:
    public:
        TContinuousSchedule(Priority, Tconstraints, Treply, TDegrade);
        virtual ~TContinuousSchedule();
        void    SetGlobal(Boolean);
        void    SetPriority(Priority);
        }

// general object to describe the constraints used by the Wilson Scheduler.
// an inherent constraint which is not listed below is that
// the source is ready to send.
// This may be changed to an abstract base class??
class TConstraints {
    private:
    public:
        TConstraints();
        virtual ~TConstraints();
        void    SetMaxDelay(Chrono del);
        void    SetMinDelay(Chrono del);
        void    SetTearFree(Boolean);
        void    SeTDoneByAbsoluteTime(Chrono time);
        void    SetMessageFrom(...);
        void    SetMessagePerFrame(...);
        // others...
    }

// general object to describe the reply formats desired.
// this may be changed to an abstract base class.
class TReply{
    private:
    public:
        TReply();
        virtual ~TReply();
        void    SetReplyDegrade(TMessage mes);      // mes to send for degradation
        void    SetReplyDone(TMessage mes);  // mes to send when done
        void    SetReplyFail(TMessage mes);  // mes to send when failed
        void    SetReplyOK(TMessage mes);           //mes to send when resource ok
        void    SetReplyFrame(TMessage mes);        // mes to send after each frame
        // others....
    }

//general object to describe degradation methods.
// This may be changed to an abstract base class.
class TDegrade{
    private:
    public:
        TDegrade();
        virtual ~TDegrade();
        void    DropFramesOk(Boolean);  //its ok to drop frames
        // degradation methods.....
    }
```

Section 4

Wilson Low Level Software

The low level software interface to Wilson is defined below. Only a single entry point is needed. The CommandChannel function is called with 4 parameters which indicate the operation to be performed. Additional parameters are passed via the the fourth parameter which is a pointer to a parameter list. Values may be returned via the parameter list. The interface defined for the motherboard is the same as the interface which will be used by expansion cards. The same interface will be used by all expansion cards desiring to support video & animation. The format of the call is:

```
ChannelCommandErr  CommandChannel(ChannelType, ChannelNo, Command, Parameters)
    Byte ChannelType;
    Byte ChannelNo;
    Byte Command;
    void *Parameters;

Error Returns (ChannelCommandErr values):
C_NOERR             No error
C_BAD_CHANNEL_TYPE  Invalid ChannelType
C_BAD_CHANNEL_NO    Invalid Channel Number for specified ChannelType
C_BAD_COMMAND       Invalid Command for specified ChannelType
C_BAD_PARAMS        Parameter list contains invalid information
C_NO_BUFFER_SPACE   Not enough buffer space to add
                       command to list (Sequence Channel only)
```

Valid ChannelTypes, Commands, and Parameter list formats

```
ChannelType=RECT_SOURCE_CHANNEL
    Command=FULL_SETUP
        short Width;
        byte BitWidth;
        short WidthCount;
        byte BitWidthCount;
        short Stride;
        byte BitStride;
        boolean Global;              cache coherency enable
        boolean Direction;
        boolean ILen;                interrupt on length =0
        boolean IPag;                interrupt on page crossing
        long Length;
        byte Priority;
        byte Stream;
        long Rate;                   16.16 megabytes per second
        long RateCount;
        InterruptVect *IntVect
        long DestAddress;
        byte StreamType;
```

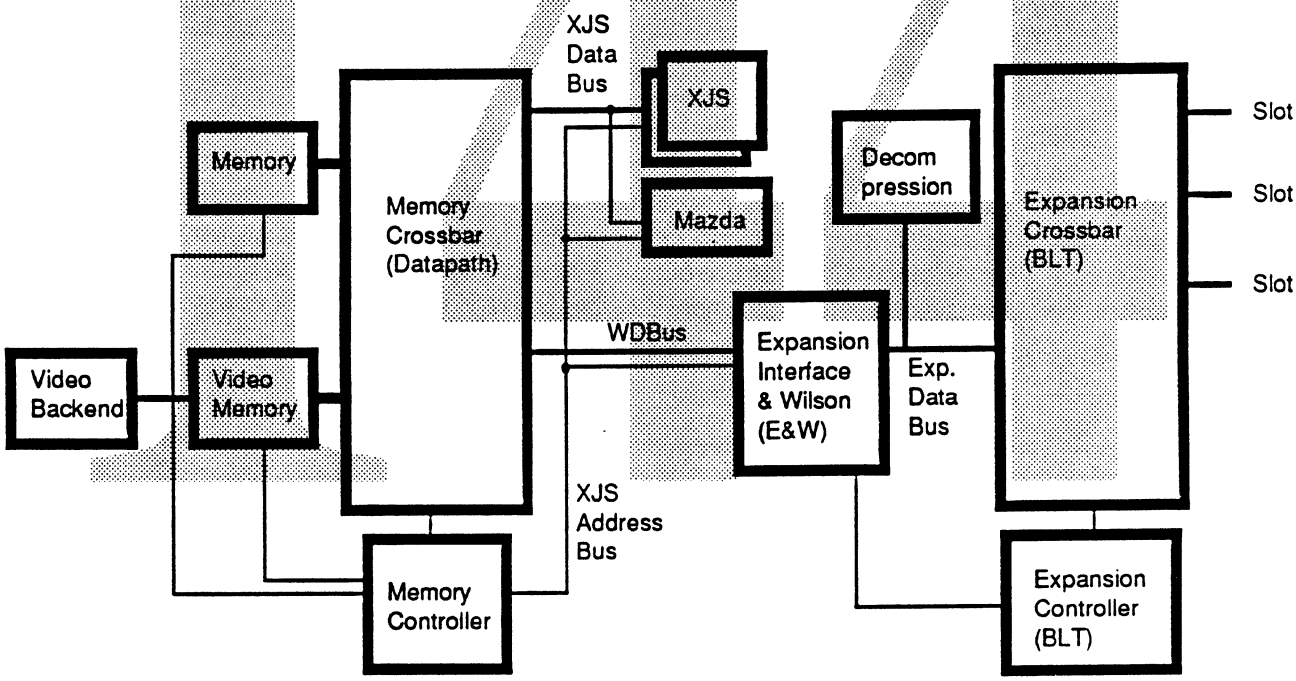
```
        boolean Constant;
        double ConstantValue;
    Coammnd=SET_PAGE
        long address;
    Command=SET_ADD
        long Address
        byte bitaddress
    Command=SET_WIDTH
        short Stride;
        byte BitStride;
        short Width;
        short BitWidth;
    Command=SET_LENGTH
        long length;
    Command=START                    starts channels transfer
    Command=STOP                     stops channels transfer
    Command=CLEAR_WIDTH_COUNT        sets current width to zero
    Command=FLUSH
    Command=READ_ERROR
        boolean Err;


ChannelType=RECT_DESTINATION_CHANNEL
    Command=FULL_SETUP
        short Width;
        short WidthCount;
        short Stride;
        boolean Direction;
        boolean ILen;                interrupt on length =0
        boolean IPag;                interrupt on page crossing
        long Length;
        byte Priority
        byte Stream;
        boolean Global;
        InterruptVect *IntVect
        boolean StoreAlpha;
        boolean StoreIntensity;
        boolean Clip
        byte StreamType;
        byte Convert;                //RGB to Y conversion & 32-64 & 16-32 conversion
    Command=SET_PAGE
        long Address;
    Command=SET_ADD
        long Address                 destination address
    Command=SET_WIDTH
        short Stride
        short Width
    Command=SET_LENGTH
        long length;
    Command=START                    starts channels transfer
    Command=STOP                     stops channels transfer
    Command=CLEAR_WIDTH_COUNT        sets current width to zero
    Command=FLUSH

ChannelType=PIXEL_MUNGER_CHANNEL
    Command=FULL_SETUP
        long DestAddress;
        byte PixelPortStreamType
        byte AlphaPortStreamType
        byte OutputStreamType
        byte CLUTNumber
        boolean double;
    Command=SET_CLUT
```

```
        byte CLUTNumber
        byte Index
        double Value
    Command=SET_FULL_CLUT
        byte CLUTNumber
        long NumberOfEntries
        double Entries[0...NumberOfEntries]
    Command=FLUSH


ChannelType=BLENDER_CHANNEL
    Command=FULL_SETUP
        long DestAddress;
        byte StreamType
        boolean AlphaOverride
        boolean PreMultiply
    Command=FLUSH


ChannelType=VIDEO_RASTER_POSITION
    Command =READ
        byte BeamNumber
        long CurrentVideoLine          current line of the video input or output.


ChannelType=SEQUENCE_CHANNEL
    Command=FULL_SETUP                 Stops command execution, clears ErrorCode
        void *BufferAddress
        Byte Priority
    Command=ADD_TO_SEQUENCE            adds a command to the command list
        void *BufferAddress            address where command will be placed
        void *BufferEndAddress         last address which can be used for the command
        void *BufferUsedAddress        return value of what was the last address used+1
        Byte ChannelType;
        Byte ChannelNo;
        Byte Command;
        void *Parameters;
    Command=INSERT_WAIT                inserts a wait for interrupt into the command list
        void *BufferAddress            address where command will be placed
        void *BufferEndAddress         last address which can be used for the command
        void *BufferUsedAddress        return value of what was the last address used+1
    Command=INSERT_INTERRUPT           inserts an interrupt generation into the command lis
        void *BufferAddress            address where command will be placed
        void *BufferEndAddress         last address which can be used for the command
        void *BufferUsedAddress        return value of what was the last address used+1
        InterruptVect *IntVect
    Command=INSERT_JUMP                inserts an absolute physical jump into command list
        void *BufferAddress            address where command will be placed
        void *BufferEndAddress         last address which can be used for the command
        void *BufferUsedAddress        return value of what was the last address used+1
        void* Address                  physical address of jump
    Command=SAVE_STATUS
        void *BufferAddress            address where command will be placed
        void *BufferEndAddress         last address which can be used for the command
        void *BufferUsedAddress        return value of what was the last address used+1
        void* Address                  physical address of status
    Command=START                      starts a channel's execution
    Command=STOP                       stops a channel's execution
    Command=STATUS
        void *BufferAddress            // instruction pointer
        Boolean Waiting
        Boolean Active
```

Section 5

Wilson Hardware Implementation

Below is a simplified overall block diagram of a Jag1 system. The E&WC is located between the Expansion Crossbar (BLT) & the Memory Crossbar. It provides an interface between the two environments.



System Diagram Showing Location of Expansion Interface & Wilson

The E&WC performs transfers between the BLT and the motherboard's local memory. The E&WC participates in the XJS cache coherency protocols on the XJS Address Bus. The data bus between the memory crossbar and E&WC, however, is a super set of a standard XJS data bus. The normal XJS address/data bus only supports 8 word, double word, single word, half word, and byte transfers. The modified data bus (the WData Bus) includes byte enables for more versatile transfers.

Hardware Operation Summary

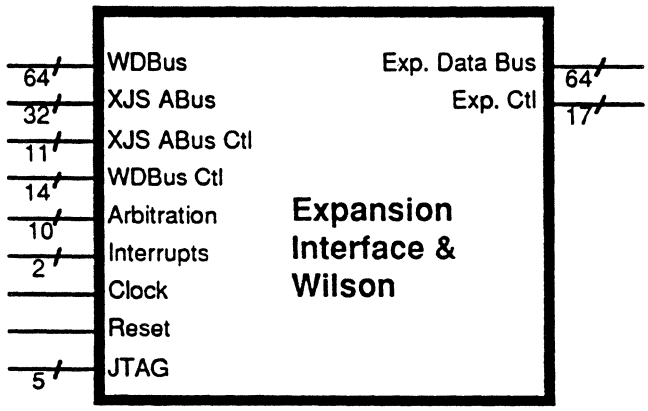
The XJS and Mazda may perform read and write operations to the E&WC and to the Expansion Crossbar (BLT). The E&WC decodes these addresses directly from the address bus. For a read operation the address is acknowledged and latched by E&WC. The source of the request is recorded by the Memory Controller. The E&WC performs the read operation via BLT or internally and returns the data to the Memory Crossbar. The Memory Crossbar delivers the data to the appropriate destination (the source of the request). For a write operation the address is latched by E&WC. The Memory Crossbar delivers the data to the E&WC. The E&WC then performs the write via BLT or internally.

The Expansion Crossbar may perform read & write operations to the system memory. If the E&WC receives a read operation from BLT it places the read request on the XJS Address Bus. The Memory Crossbar delivers the proper data on the WData Bus. The E&WC delivers the data to the destination via BLT. If the E&WC receives a write operation from BLT it places the write request on the XJS Address Bus and passes the data to the Memory Crossbar via the WData Bus. The Memory Crossbar delivers the data to the proper destination.

Inside of the E&WC is the logic for performing Wilson Transfers. The E&WC may perform read and write operation via BLT or the Memory Crossbar to accomplish Wilson Transfers. Reads may be performed from the Memory Crossbar and the data written to a destination via BLT (transfer to an external frame buffer). Reads may be performed from the Memory Crossbar and the data written back to the Memory Crossbar (bit-blit). Wilson can also receive writes from BLT which it translates before passing them to the Memory Crossbar (a live video window for example).

Hardware Interface

Pinout



Expansion Interface & Wilson Pinout

Group	Full Name	Abbrev. Name	#of pins	Directions
XJS ABus	Address Bus	A31-0	32	bidirectional
XJS ABus Ctl	Transfer Size	TSIZ1-0	2	bidirectional
	Transfer Burst	TBST*	1	bidirectional
	Read/Write	R/W*	1	bidirectional
	Lock	LK*	1	bidirectional
	Transfer Start	TS*	1	bidirectional
	Address Acknowledge	AACK*	1	input
	Intent to Modify	IM*	1	output
	Memory Cycle	MC*	1	output
	Snoop Retry	SRTRY*	1	input
	Global	GBL*	1	output
WDBus	Wilson Data Bus	WD63-0	64	bidirectional
WDBus Ctl	Transfer Acknowledge	WTA*	1	input
	Transfer Error Acknowledge	WTEA*	1	input
	Transfer Retry	WTRTRY*	0[1]	input
	Byte Enables	WBE7-0*	8	output
	Slave Transfer Acknowledge	SlvWTA*	1	output
	Slave Transfer Error Acknowledge	SlvWTEA*	1	output
	Slave Transfer Retry	SlvWTRTRY*	1	output
Arbitration	Bus Request Priority 0	BR03*	1	output
	Bus Request Priority > 0	BR13*	1	output
	Bus Grant (address bus)	BG3*	1	input
	Address Bus Busy	ABB*	1	bidirectional
	Data Bus Busy	WDBB*	1	bidirectional
	Data Bus Grant	WDBG*	1	input
	Wilson Slave Select	SlvWSEL*	1	input
	Wilson Slave Write Ready	SlvWWR*	1	output
	Wilson Slave Read Ready	SlvWRR*	1	output
	Wilson Slave Address Ready	SlvWAR*	1	output
Miscellaneous	Interrupt0-1	WINT0-1*	2	output
	Clock	WCLK	1	input
	Reset	RESET	1	input
	JTAG4-0	JTAG4-0	5	bidirectional
Exp. Data Bus	Exp. Data Bus	IOD31-0	32	bidirectional
	Exp. Data/Address Bus	IOD63-32	32	bidirectional
Exp. Ctl	Twist	TWT2-0	0[2]	output
	Size	SIZ2-0	3	bidirectional

[1]Transfer Retry is not needed between E&W and the Memory System when the E&W is acting as a master. The Memory system will never indicate a retry to the E&W.

[2] If the #of Pins=0 the pins exist on the BLT interface but are not connected to the E&W.

	Lock	LK*	1	bidirectional
	Data Output Enables	DOE1-0*	2	output
	Acknowledge	ACK*	1	output
	Node	NODE3-0	(4)[1]	bidirectional
	Priority Select	PSEL	(1)	bidirectional
	Priority/Stream	PRST2-0	(3)	bidirectional
	Packet Type	PTYPE3-0	(4)	bidirectional
	Header Valid	HVAL*	1	bidirectional
	Header Output Enable	HOE*	1	output
	Done	DON*	1	output
	Skip	SK*	1	output
	Complete	COM*	0	input
	Packet Here	PHER*	1	input
	Send Buffer Available	SBA*	1	input
	Abort	AB*	0	input
	Stream Full	STF*	1	input
	Error	ERR*	1	input
	Error Output	ERRO*	1	output
	Clock	CLK	0	
	Interrupt Request	IRQ*	0	
	SysClock	SCLK	0	
	Reset	RESET*	0	
	Delayed Error	DERR*	1	input
	Low Power	LPOW*	0	
TOTAL			220	
Power/Gnd			64	
GRAND TOTAL		284		

Pin Descriptions

XJS ABus

See the XJS Specification for details

XJS ABus Ctl

See the XJS Specification for details

WDBus

WD63-0 -- The data lines are used to transfer data to and from the Memory Crossbar.

[1] The numbers in parenthesis are not actual pins on the E&W chip. These pins are multiplexed with the I/O Data Bus pins on the E&W.

WDBus Ctl

Transfer Acknowledge -- See the XJS Specification for details

Transfer Error Acknowledge -- See the XJS Specification for details

Transfer Retry -- See the XJS Specification for details

Byte Enables -- When data is transferred from the E&WC to the Memory Crossbar these lines indicate which bytes are valid. These lines are valid for all writes from the E&WC to the Memory Crossbar.

Slave Transfer Acknowledge -- Same as Transfer Acknowledge accept used when E&WC is a slave device.

Slave Transfer Error Acknowledge -- Same as Transfer Error Acknowledge accept used when E&WC is a slave device.

Slave Transfer Retry -- Same as Transfer Retry accept used when E&WC is a slave device.

Arbitration

Bus Request Priority 0 -- Asserted when the E&WC desires to perform a transaction at priority zero.

Bus Request Priority > 0 -- Asserted when the E&WC desires to perform a transaction at a priority greater than zero.

Bus Grant (address bus) -- See the XJS Specification for details

Address Bus Busy -- See the XJS Specification for details

Data Bus Busy -- See the XJS Specification for details

Data Bus Grant -- See the XJS Specification for details

Slave Select -- Asserted by the Memory Controller to indicate to the E&WC that data is valid for a write or that data is desired for a read. Slave Select is not asserted unless one or both the Slave Ready lines are asserted. If both Slave Read Ready and Slave Write Ready are asserted the read transaction will be performed first.

Wilson Slave Write Ready -- Asserted by the E&WC to indicate that it can accept a write transaction. The Memory Controller will not attempt to send data for a write operation to the E&WC if this line is not asserted.

Wilson Slave Read Ready -- Asserted by the E&WC to indicate that read data is available for a reply. The Memory Controller should assert Slave Select to obtain the data.

Wilson Slave Address Ready -- Asserted by the E&WC to indicate that it has an address buffer available for a slave transaction. The Memory Controller will not driver **AACK*** if the transaction is destined for the E&WC and this line is not asserted.

Miscellaneous

Interrupt0-1 -- Outputs. Driven low to interrupt one or the other XJS CPU's. These lines are connected to the Mazda I/O Chip.

Clock – Input. Maximum 50 MHz clock input. Connected to internal phase lock loop for clock skew minimization.

Reset -- Input. Driven low at power up and during sleep mode. Does not clear any state registers. Places part into lowest power consumption mode possible.

JTAG4-0 -- Bidirectional. Defined by ASIC vendor.

Exp. Data Bus

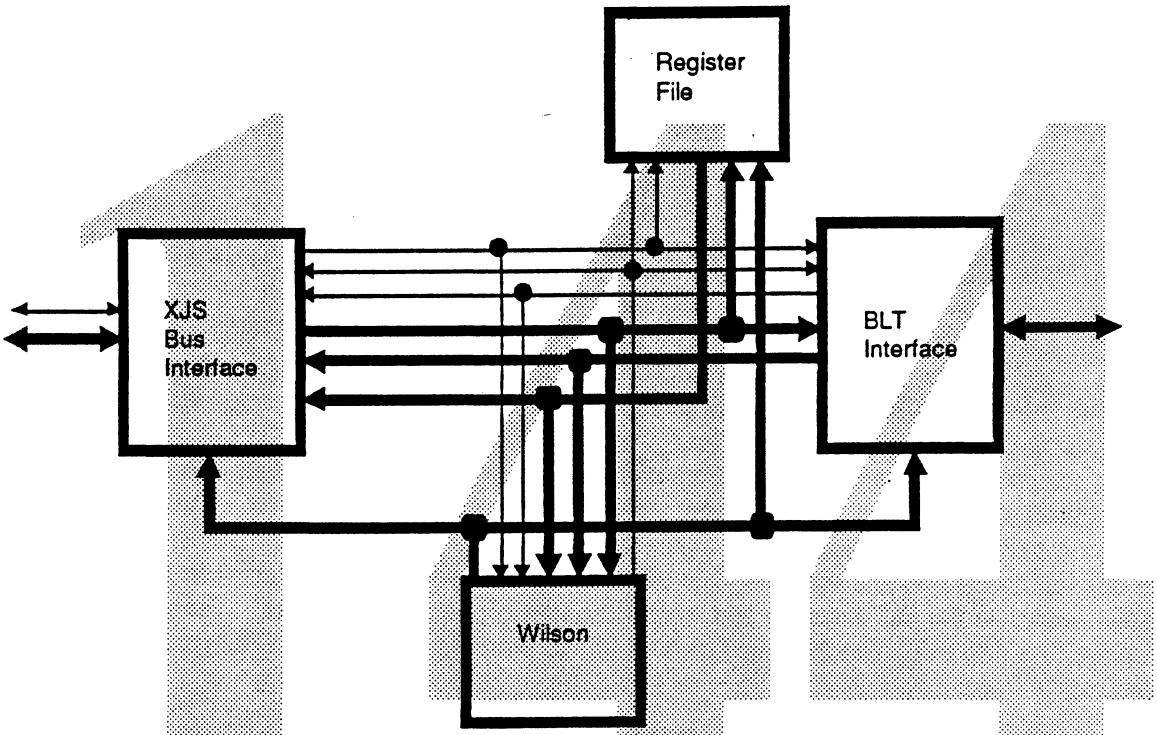
See the BLT ERS for details.

Exp. Ctl

See the BLT ERS for details. Note that the Node3-0, Priority/Stream2-0, Priority Select, and Packet Type3-0 are multiplexed with the Exp. Data Bus11-0.

Implementation Description

An overall internal block diagram of the E&WC is shown below. The thick lines represent data paths (nominally 64 bits). The thin lines represent address paths.



Expansion Interface & Wilson Internal Block Diagram

The XJS Bus Interface Block connects to the XJS Address Bus and the WData Bus. It performs both master and slave read and write operations of up to 8 words. The BLT Interface connects the E&WC to the Expansion Crossbar (BLT). It performs master reads and writes of up to 8 words and slave reads and writes of up 16 words. The BLT Interface Block and the XJS Bus Interface Block communicate to perform standard transactions between the two environments.

The Wilson Block contains the DMA logic for moving graphic regions. It is basically a complex DMA engine designed to support BLT. The Wilson Block consists of two fundamental halves: 1) the source half (RectRegion Source Resource), and 2) the destination half (RectRegion Destination Resource). The RectRegion Source Resource can read memory (via the XJS interface or the BLT Interface) and can write the data to either the RectRegion Destination Resource, the XJS Bus Interface, or the BLT Interface. The RectRegion Destination Resource is written to by the RectRegion Source Resource, the XJS Bus Interface, or the BLT Interface. It may transform the data (change it from 24 bits per pixel to 8 bits per pixel for example) but will write the data to either the XJS Bus Interface or the BLT Interface. The Register File Block may be written to and or read by the XJS Bus Interface. It provides access to all registers contained in the E&WC. The Wilson Block may

also read or write the Register File. The BLT Interface Block can access the internal registers via the XJS Interface Block (through and back loop).

The Register File Block provides read and write access to internal registers on the E&WC. The registers are not necessarily located in the Register File Block. The Register File Block, however, provides read and write access to the registers.

Implementation Details

BLT Interface

BLT Slave Read -- Slave read operations are requested by BLT. The BLT signals a packet has arrived. The BLT Interface decodes the packet as a Read Packet. The address, size, priority, and source are latched from BLT. The BLT Read Packet is acknowledged. The address, priority, and size are placed on the internal BLT Address Bus. A read request is signaled to the XJS Bus Interface. The XJS Bus Interface acknowledges the address and read request. The BLT Interface waits for the data to be returned by the XJS Bus Interface. The XJS Bus Interface places the data on the data connection between the two Interfaces. The XJS Bus interface signals that the data is a read reply. The BLT Interface stores the incoming data into an 8 double word entry buffer. The BLT Interface sends a header to the BLT Interface and signals the start of a read reply packet. The BLT Interface sends the data into BLT. The BLT Interface tells BLT to send the reply packet. The read operation is complete.

Two read request can be serviced at one time. The XJS Interface is responsible for obtaining the requested data and returning it to the BLT Interface Block. Note that the size may be up to 64 bytes and that it must be tightly packed. The data may or may not be received in a single burst. Multiple bursts may be needed or used by the XJS Interface. The XJS Bus Interface must guarantee order.

BLT Master Read -- Master read operations are requested by the XJS Bus Interface or the Wilson Block. Read requests can be received from two different sources. If two requests are received simultaneously the BLT Interface Block only acknowledges one of them. The address is decoded and acknowledged only if it is located on BLT. The BLT Interface Block latches the address, priority, size, and source of the read request. It acknowledges the address to the requester. The BLT Interface Block arbitrates for the BLT Bus and sends the read request packet (address, priority, destination node, size). It then waits for BLT to return the data. When the data is returned the BLT Interface clocks the data from the source and returns it to the requester via its output data bus. The data is returned compressed into 64 bit words.

The BLT interface can have two outstanding read operations. It identifies the reply using the Node and Priority bits. Only reads of up to 4 words double words are serviced.

BLT Slave Write – Slave write operations are requested by BLT. They may be executed to either the Wilson Block or the XJS Bus Interface. If the Write is not a stream write the address and data for the write are acknowledged. If the write is a stream write the destination is checked to see

if it can receive a packet. If it can the address and data are acknowledged. If it cannot the write packet is "Skipped". Once data is acknowledged it is either forwarded to the destination or buffered in a local buffer. When the buffer is full further BLT writes will not be acknowledged until the buffer empties. The address of the write is placed on the BLT Interfaces outgoing (internal) address bus. Data is placed on the data bus. See the BLT section on stream support for further details.

BLT Master Write -- Master write operations are requested by the XJS Bus Interface or the Wilson Block. Requests are received on the address lines. The address is decoded by the BLT Interface Block. The BLT Interface acknowledges the address if its buffer is empty. It immediately begins setting up BLT to receive a write packet. The packet is received by the BLT Interface and sent to BLT. If BLT cannot accept the data immediately the BLT Interface places it into an internal buffer. When the buffer is full the BLT Interface will not acknowledge a write request.

XJS Bus Interface

XJS Slave Read -- Slave read operations may be requested by devices located on the XJS Address Bus. The XJS Interface Block observes the XJS Address bus. Whenever it decodes an address which is either located internal to the E&WC or located out on BLT it latches the address. It can queue two such read requests before the address bus is not acknowledged (**SLVWAR*** is driven inactive). It sends the requested read (the address may have to be modified to account for the CPU's cache line wrap if the operation was a cache line burst read). The address is acknowledged by the destination which decodes it. The XJS Bus Interface Block then waits for the data. The XJS Bus Interface Block acknowledges the data and places it into a 4 double word entry buffer. The XJS Bus Interface Block arbitrates for the external local data bus. It sends the data back to the Memory Crossbar indicating that it is a read response. The word order may have to be changed if the request was a cache line read (Critical word first). If the read operation was a partial word (1,2, or 4 bytes) the byte lane(s) may have to be changed. Misaligned transfers are not supported.

XJS Master Read -- Master read operations may be requested by the Wilson Block or the BLT Interface. The XJS Bus Interface Block latches the address, priority, size, and source of the read request if the address is in the Local Memory address space. Up to 3 requests can be queued and/or launched onto the XJS address bus. The XJS Bus Interface Block arbitrates for the external XJS Address Bus. The addresses (requests) are launched on the XJS address bus. The addresses may be discarded after they have been sent on the XJS address bus. The Memory Crossbar eventually fetches the data and requests the Local Data Bus for a read reply. The data is transferred through the XJS Bus Interface Block and into the original requester's buffer. The XJS Bus Interface Block discards any unneeded data (possible obtained due to wrapped reads) and forwards any remaining data to the requester who is waiting. The data might be twisted by the XJS Bus Interface Block to place it onto the low order byte lanes. If the read size requested still has not been fulfilled the XJS Bus Interface Block increments the address properly and launches another read onto the address bus (it may have already actually done this). The XJS Bus Interface Block is responsible for making multiple XJS Bus transactions appear as single transactions to the requesters. If a misaligned block read operation is performed the XJS Bus Interface Block will perform the necessary transactions on the XJS bus and compress the data into 64 bit words. Misaligned requests are supported. Data returned from the

memory crossbar is guaranteed to maintain order. Data returned by the XJS Bus Interface Block is also guaranteed to maintain order.

XJS Slave Write -- Slave write operations may be requested by devices located on the XJS Address Bus. The XJS Bus Interface decodes all address on the XJS Address Bus. When the address is in the BLT or Wilson Address Space and the XJS Bus Interface has room in its address buffers it latches the address. Up to 3 addresses for write operations may be buffered before the XJS address bus is blocked. Eventually the data for the write operations is delivered to the E&WC by the Memory Crossbar. The data is labeled as write data and is guaranteed to maintain order. The XJS Bus Interface buffers the data and attempts to send it sequentially to the destination. The address and data are placed on the internal address and data busses from the XJS Bus Interface. The address and data are acknowledged by the destination (Wilson, Register File, or BLT Interface).

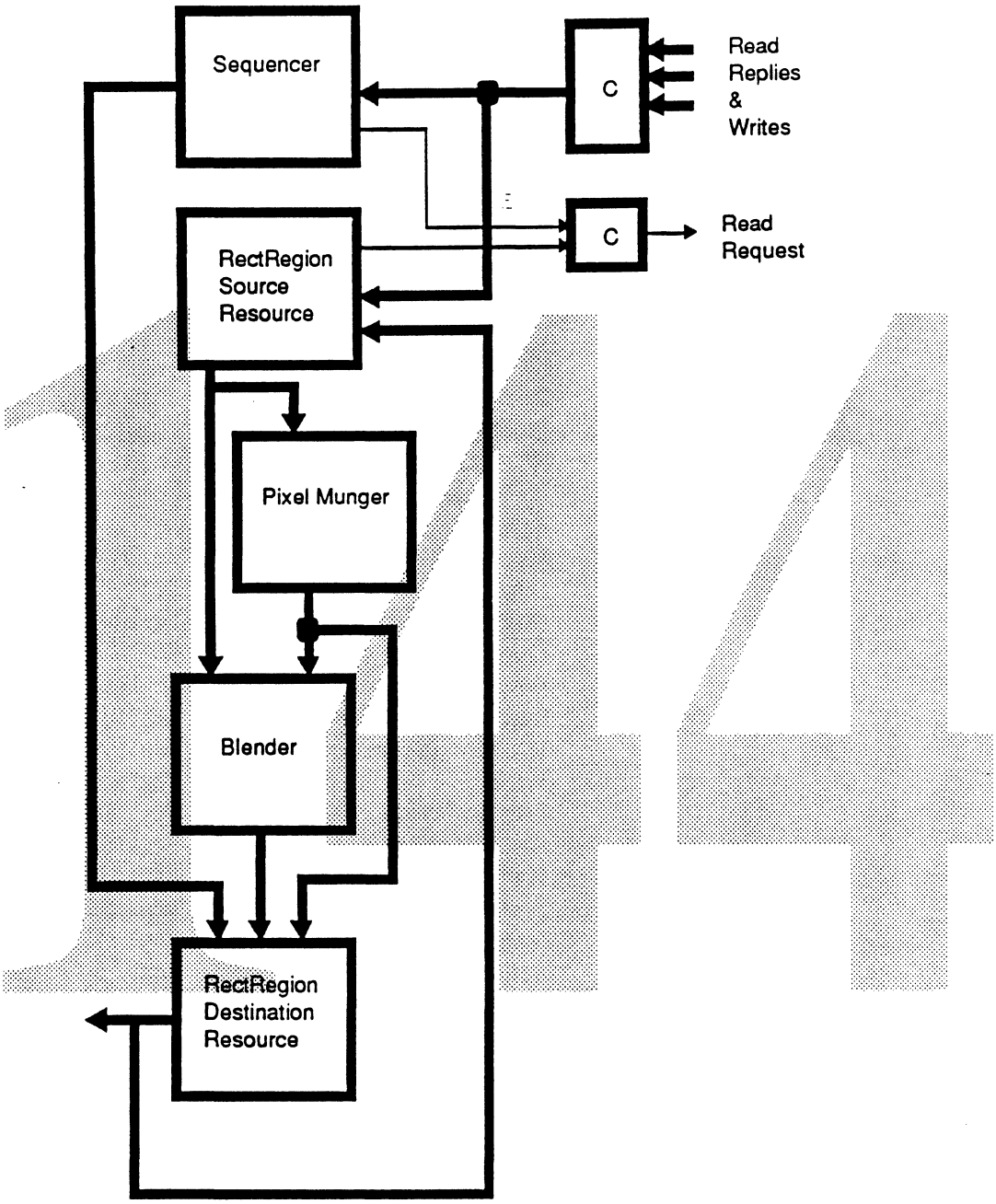
XJS Master Write – Master write operations may be requested by the Wilson Block or the BLT Interface. Write transactions can be received on two address and data busses. If two transactions are received simultaneously only one is acknowledged. The XJS Bus Interface attempts to stream the transactions out to the E&WC Data Bus without buffering but provides buffering in case the E&WC Data Bus is busy. Once the first word of a packet is acknowledged all the remaining words will be accepted at full speed. The XJS Bus Interface arbitrates for the external XJS Address Bus. The write transaction is launched on the XJS Address Bus and is acknowledged by the Memory Controller. The XJS Bus Interface then sends the data to the Memory Crossbar. The Memory Crossbar delivers the data to the proper destination based on the address.

Register File

The state registers on the E&WC can be read and written by both the Wilson Block and the XJS Bus Interface Block. The Register File Block provides access to all the internal registers which are visible. If two transactions are simultaneously requested only one is acknowledged. These registers are not generally cleared with a Reset (some of the one's indicating activity are).

Wilson

An internal diagram of the Wilson Block is shown in the figure below. It consists of five sub-blocks. The sequencer performs sequencing operations of the other channels. This is used for performing scatter/gather operations in the virtual memory environment and for setting up and controlling execution queues. The RectRegion Source Resource reads memory and creates streams. The Pixel Munger accepts two streams and performs multiplexing and selecting operations to create a single output stream. The Blender accepts two streams and performs an alpha based blend operation to create a single output stream. The RectRegion Destination Resource translates a stream into memory writes and performs alpha based clip operations. The Blender and the Pixel Munger are Stream Processing Resources.



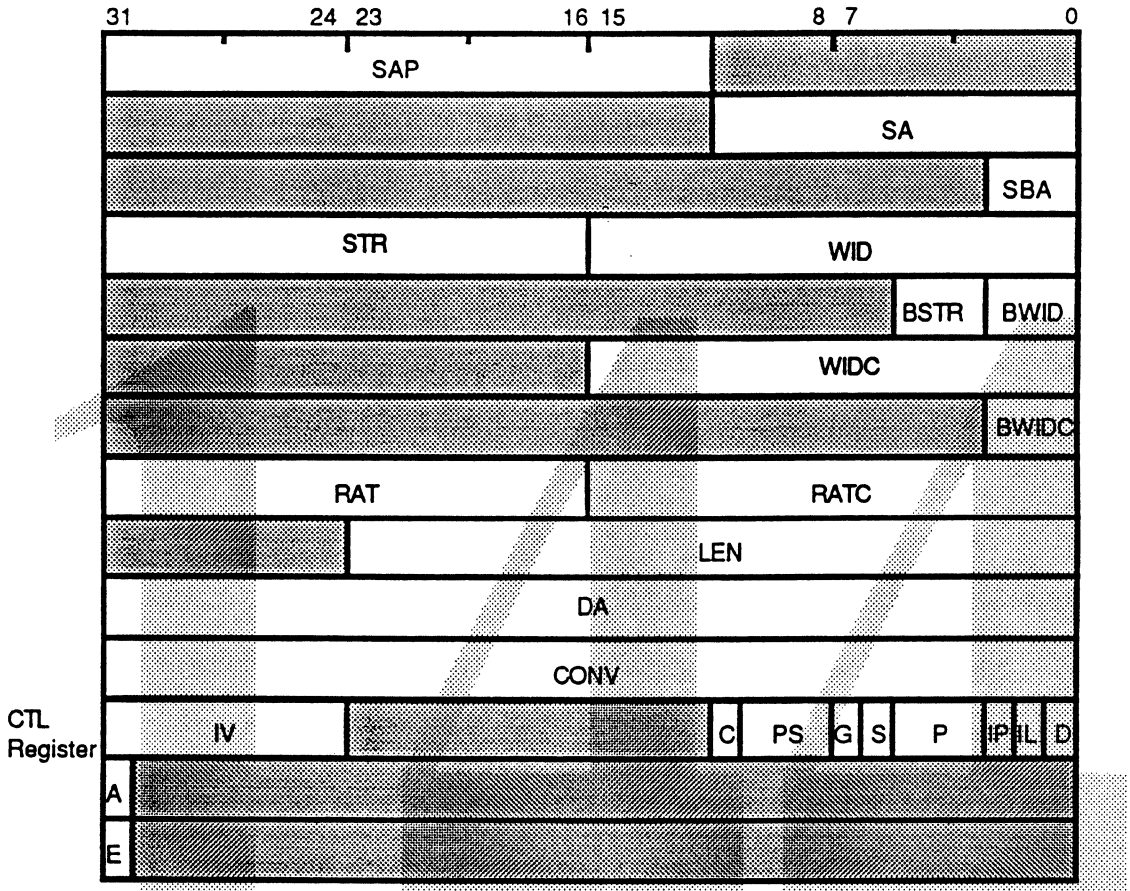
Internal Diagram of Wilson Block

RectRegion Source Resource (RRSR)

The motherboard's RectRegion Source Resource (RSR) is the DMA read engine of the E&WC. It contains 8 channels capable of reading rectangular region structures from memory. Each channel operates independently. Data is read from memory and passed on to the Pixel Munger, Blender, RectRegion Destination Resource, or to an external destination. Writes which are received from

either the XJS Bus Interface or the BLT Interface are also routed through the RectRegion Source Resource. The write's destinations can be either the Pixel Munger, the Blender, or the RectRegion Destination Resource. The RectRegion Source Resource contains the following state information for each channel:

Register Name	Bits	Descriptive Name	Location
SAP	20	**Source Address Page**	Reg0[31:12]
SA	12	**Source Address**	Reg1[11:0]
SBA	3	**Source Bit Address**	Reg2[2:0]
WID	16	**Width**	Reg3[15:0]
BWID	3	**Bit Width**	Reg4[2:0]
STR	16	**Stride**	Reg3[31:16]
BSTR	3	**Bit Stride**	Reg4[5:3]
WIDC	16	**Width Count**	Reg5[15:0]
BWIDC	3	**Bit Width Count**	Reg6[2:0]
RAT	16	**Rate**	Reg7[31:16]
RATC	16	**Rate Count**	Reg7[15:0]
LEN	24	**Length**	Reg8[23:0]
DA	32	**Destination address**	Reg9[31:0]
CONV	32	**Constant Value**	Reg10[31:0]
D	1	**Direction**	Reg11[0]
IL	1	**Interrupt on Length**	Reg11[1]
IP	1	**Interrupt on Page Boundary**	Reg11[2]
P	3	**Priority**	Reg11[5:3]
S	1	**Priority/Stream**	Reg11[6]
G	1	**Global**	Reg11[7]
PS	3	**Pixel Size**	Reg11[10:8]
C	1	**Constant**	Reg11[11]
IV	8	**Interrupt Vector**	Reg11[31:24]
A	1	**Active**	Reg12[31]
E	1	**Error**	Reg13[31]



RectRegion Source Resource Channel Registers

All registers are readable and writable by the System CPU. Some of the registers may be changed by the Channel itself. None of the registers should be changed if the **Active** bit is one.

The **Source Address Page** is the physical page number which will be used for the source data. The **Source Address** is the byte address within the page which will be used for the source data. The RectRegion Source Resource Channel will read data from the address indicated by the **Source Address Page** and the **Source Address.** The **Source Address** is incremented for every byte read. If it rolls over to zero the **Source Address Page** is incremented. If the **Interrupt on Page Boundary** bit is set the channel will generate an interrupt (and stop data movement) when the **Source Address Page** is incremented. The **Source Bit Address** is used in conjunction with **Source Address** and **Source Page Address** when the pixel size is one bit (generally one bit mask data) to obtain a full bit address.

The **Width** and the **Width Count** are control the width of the rectangular region being moved. For every byte read the **Width Count** is decremented. When the **Width Count** reaches zero the sign extended **Stride** is added to the **Source Address** and **Source Address Page**. The **Width Count** is also reloaded from the **Width** register at this time. Future data will be read from the new address. The **Bit Stride**, **Bit Width**, and **Bit Width Count** are used when the pixel size is one bit.

The **Width Count** Register must be loaded at the beginning of the transfer. The **Width Count** is not automatically loaded at the beginning of a transfer. It is only reloaded when it reaches zero. This allows a virtual page to be changed without affecting the region transfer.

The **Rate** and **Rate Count** are used to control the rate of the transfer. The **Rate Count** is decremented at a fixed rate of 1 MHz. A packet is read whenever the **Rate Count** reaches zero. At the same time the **Rate** value is transferred into the **Rate Count**. If the **Rate** is set to zero the channel will operate at maximum speed.

The **Length** register is decremented for each byte read from memory. When the **Length** register reaches zero the channel stops execution. If the **Interrupt on Length** bit is set the channel will generate an interrupt when the **Length** reaches zero.

The **Destination Address** is the location where all data is written. It is never incremented or changed by the channel.

The **Constant Value** register contains a 32 bit constant. A stream of a constant value can be source by the channel if the **Constant** bit equals one. The value of the stream is taken from the **Constant Value** register.

The **Interrupt Vector** is used to generate an interrupt when the channel stops execution. This can either occur because the **Length** register reached zero or because a page boundary was crossed. See the interrupt section for details.

The **Direction** bit controls whether the **Source Address**, **Source Page Address**, and **Source Bit Address are** incremented or decremented. If the bit is set (Forward) the address is incremented for each byte transferred. If the bit is clear (reverse) the address is decremented for each byte transferred. When reverse is set byte lanes are swapped as necessary to create a completely backwards pixel stream.

The **Pixel Size** bits controls the size of the pixels being read from memory. The following sizes are supported: 0) 1 bit, 1) 1 byte, 2) 2 bytes, 3) 3 bytes, 4) 4 bytes. If the **Pixel Size=0** the **Source Bit Address, Bit Width, Bit Width Count,** and **Bit Stride** are used to obtain full bit addresses for the region. The data is read in a bit fashion and expanded on output to a full byte. This provides one bit masks for clipping. Bit replication is used to expand from one bit to 8 bits.

The **Constant** bit allows a constant to be substituted for the output stream. If **Constant=1** the 32 bit value in the **Constant Value** register is used. This value is repeatedly sent to the destination until the **Length** reaches zero.

The **Priority** and **Global** bits are used when performing read and write operations. If **Global=1** all reads and writes are marked as global. All transactions are executed at the priority indicated in the **Priority** register.

The **Priority/Stream** bit controls the operation of data sent via BLT. The data can be labeled with a particular stream number if the stream bit is set.

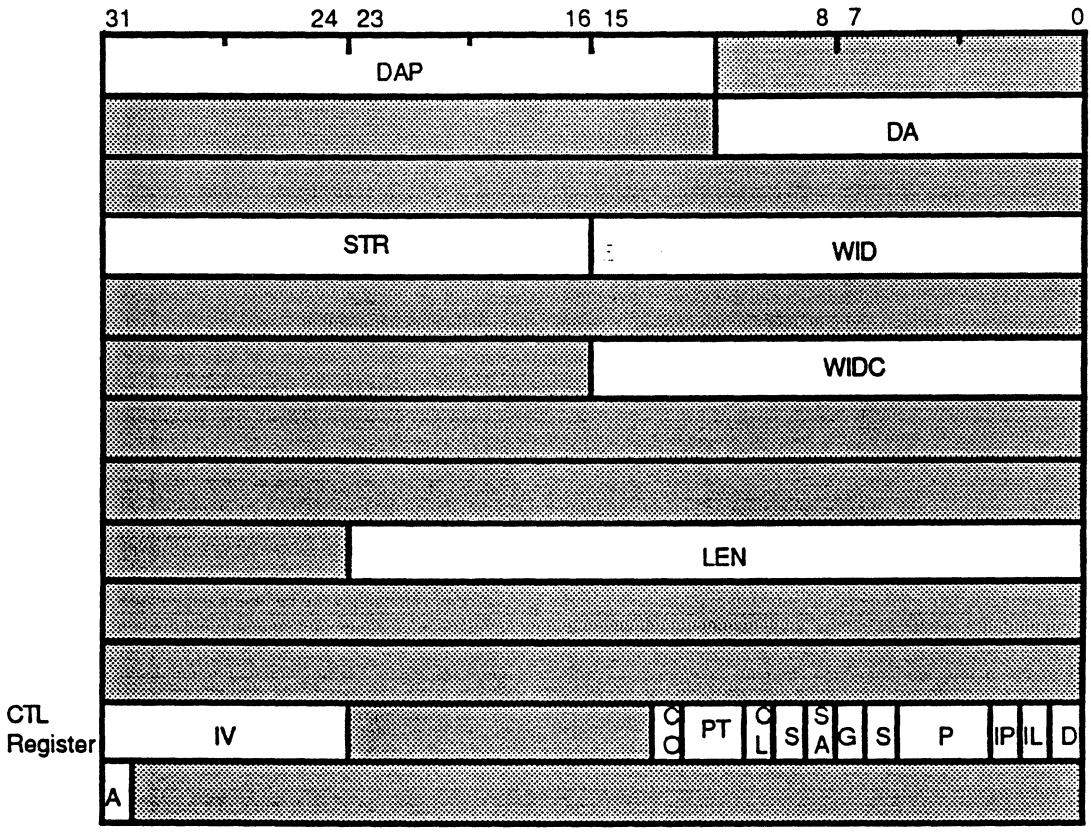
The **Active** bit controls the whether the channel operates or not. If **Active=1** the channel will fetch data from memory. **Active** is set to zero when an interrupt is generated.

The **Error** bit is set when an read error occurs. The channel immediately stops execution and generates an Error Interrupt. See the error section for details. The **Error** bit must be cleared by a write to the register.

RectRegion Destination Resource (RRDR)

The RectRegion Destination Resource receives input streams from the Blender, the Pixel Munger, the RectRegion Source Resource, and external sources. It attaches an address to the stream of data and writes it to memory. It contains the following state for each channel:

Register Name	Bits	Descriptive Name	Location
DAP	20	**Destination Address Page**	Reg0[31:12]
DA	12	**Destination Address**	Reg1[11:0]
WID	16	**Width**	Reg3[15:0]
STR	16	**Stride**	Reg3[31:16]
WIDC	16	**Width Count**	Reg5[15:0]
LEN	24	**Length**	Reg8[23:0]
D	1	**Direction**	Reg11[0]
IL	1	**Interrupt on Length**	Reg11[1]
IP	1	**Interrupt on Page Boundary**	Reg11[2]
P	3	**Priority**	Reg11[5:3]
S	1	**Priority/Stream**	Reg11[6]
G	1	**Global**	Reg11[7]
SA	1	**Store Alpha**	Reg11[8]
SI	1	**Store Intensity**	Reg11[8]
CL	1	**Clip**	Reg11[10]
PT	2	**Pixel Type**	Reg11[12:11]
CO	1	**Convert**	Reg11[13]
IV	8	**Interrupt Vector**	Reg11[31:24]
A	1	**Active**	Reg12[31]



RectRegion Destination Resource Channel Registers

The state parameters function the same as for the RectRegion Source Resource. Some additional parameters have been added. The **Store Alpha** bit controls whether alpha information is written into the destination memory. If **Store Alpha=1** alpha data is written. If **Store Alpha=0** alpha information is eliminated from the data stream. **Store Intensity** functions in the same manner as **Store Alpha** except it affect the intensity data. **Clip** affects the alpha clip function of the Destination Resource. If **Clip=1** pixels which have an alpha=0 will not be written to memory. **Pixel Type** informs the RectRegion Destination Resource of the type of pixels which are being received (16_GRAY, 32_RGB, 32_GRAY, or 64_RGB). **Convert** controls the conversion between RGB and intensity. If the **Convert=1** pixel data will be converted to the opposite format (RGB to Y or Y to RGB) before being written. When converting from Y to RGB the intensity is replicated for all three components. When converting from RGB to Y the equation Y=9/32*R + 19/32*G + 4/32*B is used.

Pixel Munger

The pixel munger is a Stream Processing Resource located on the motherboard of Jag1 in the E&WC. The pixel munger accepts one or two input data streams and generates a single output data stream. The Pixel Munger consists primarily of multiplexers. The two streams can be of any of the following forms:

```
k16_GRAY
```

```
k16_PSEUDO
k32_RGB
k8_ALPHA
k8_GRAY_NO_ALPHA
k8_PSEUDO_NO_ALPHA
k24_RGB_NO_ALPHA
```

The #1 input stream must contain intensity data. The #2 input stream must contain alpha data. The output is always either `k16_GRAY` or `k32_RGB`. The pixel value is extracted from the first source stream and combined with the alpha from the second source stream to create the output stream. If the pixel data stream is labeled as Pseudo Color it is converted to 24 Bit RGB via a color look up table. Only one color look up table exists in hardware for all the Pixel Munger channels. The Pixel Munger contains the following state for each channel:

Register Name	Bits	Descriptive Name	Location
DA	32	**Destination Address**	Reg0[31:0]
S1FRM	3	**Stream #1 Format**	Reg1[2:0]
S2FRM	3	**Stream #2 Format**	Reg1[5:3]
S1ONLY	1	**Stream #1 Only**	Reg1[6]
DBL	1	**Double each Pixel**	Reg1[7]

The **Destination Address** is the location where the data will be written. The two **Stream Formats** describe the format of the input streams. The format of the output stream is determined by the format of the input streams. The **Stream #1 Only** bit controls the number of input streams. If the bit is set only one input stream will be used to generate the output stream. This is only useful for performing a `k16_PSEUDO` to `k32_RGB` conversion. If the **Double each Pixel** bit is set each output pixel will be sent twice in the output stream. This is used (along with a Sequencer) to perform region size doubling.

Blender

The Blender is a stream processing resource. The Blender accepts one or two input data streams and generates a single output data stream. The two streams must both be either `k16_GRAY` or `k32_RGB`. The Blender can perform three operations: 1) blend two streams, 2) blend two streams with alpha override, 3) premultiply a stream.

Register Name	Bits	Descriptive Name	Location
DA	32	**Destination Address**	Reg0[31:0]
AO	1	**Alpha Override**	Reg1[0]
PREM	1	**Premultiply**	Reg1[1]

The **Destination Address** is the location where the data will be written. The second streams pixel values are multiplied by 1-alpha of the first stream and added to the pixel values of the first stream. The second streams alpha is treated like the other pixel information unless the **Alpha Override=1**. When **Alpha Override=1** and the second streams alpha value=0 it forces the output alpha value to 0. This is used for maintaining Clip Alpha when compositing.

When **Premultiply=1** the second input stream is ignored. It is not needed. Each of the pixel values is multiplied by the stream's alpha value.

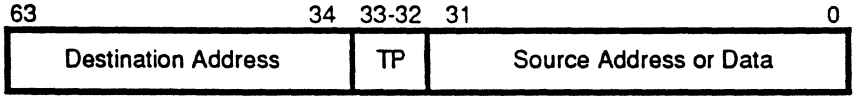
A problem exists when trying to represent an alpha value in only 8 bits (1 is represented by 255 and 0 is represented by 0) and one desires to perform only 8 bit multiplies and adds. The basic problem is that if you multiply 255*255 and truncate or round you obtain 254. 254 is not an acceptable answer because you have reduced the range which may be obtained. The solution is to special case alpha slightly and increase its range to 0 to 256. In increasing the range of alpha a "hole" must be placed in the alpha range. The Blender will place this hole at alpha=255. When alpha=255 it will be incremented to 256 before the multiply is performed.

Sequencer Block

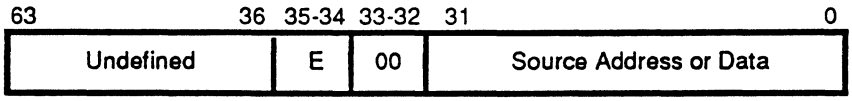
The Sequencer is a simple block which executes instructions from main memory. The Sequencer Block can perform only one basic operation: data movement. It can move a 32 bit word of data from one address location to another. The sequencer block has several channels. Each channel's state is shown below:

Register Name	Bits	Descriptive Name	Location
IP	32 (28)	**Instruction Pointer**	Reg0[31:0]
P	3	**Priority**	Reg1[2:0]
ACT	1	**Active**	Reg2[0]
WAIT	1	**Waiting**	Reg2[1]
ERR	1	**Error**	Reg3[0]

The **Instruction Pointer** points to the address where the channel will fetch instructions. There is only one functional Sequencer Block which is multiplexed between the sequencer channels. All instructions are 64 bits and must be double word aligned. The **Active** bit controls the channels operation. If **Active=0** the channel does not operate. **Waiting** is an indicator/state bit. When **Waiting=1** the channel is waiting for an interrupt before proceeding. It will not execute the next instruction until an interrupt is received. The **Waiting** bit is set when a wait instruction is executed. The **Waiting** bit is cleared when an interrupt is received.



Instruction Format



Extended Instruction Format

The first word of the instruction contains a destination address. The lower 2 bits of the destination address encode the instruction type (TP). The four types are 11) Move, 10) Move Immediate, 01) reserved, 00) Extended Instruction. If the instruction type is a Move the second word of the instruction contains a 32 bit source address. Data will be moved from the source address to the destination address. If the instruction type is a Move Immediate the second word of the instruction contains 32 bits of data. The data is moved to the destination address. If the instruction type is Extended then further decoding is necessary. Three possible extended instructions are possible which are encoded using bits 35 and 34. 00) Halt, 01) Jump, and 10) Wait. 11) is reserved. The Halt instruction causes the channel to stop operation. The Wait instruction causes the channel to stop operation until an interrupt is received. The Jump instruction loads the Instruction Pointer with the value in the data word.

Writes to the **Instruction Pointer** are ignored if the Channel is active. This provides both a safety mechanism and a queuing mechanism. The queuing method is explained in the Programmers Model section. Writes to the **Instruction Pointer** register also change the **Active** bit to a one.

Arbitration

Arbitration for the XJS Address Bus & the memory banks is priority based. It is completely fair at a given priority & completely unfair (or as near as possible) between priorities. The CPU's always operate at priority 0 (lowest priority). The E&WC operates at either priority 0 or priority 1. When performing transfers at priority 1 the E&WC could block the CPU for an extended period of time. This is guaranteed not to occur by the E&WC due to rate control (all Wilson Sources have rate control). The Memory Controller is responsible for arbitration of the XJS Address Bus and the memory modules.

The priority of each Source Resource Channel in the E&WC is programmable. The Channel operates at the programmed priority when performing reads from main memory. The Channel may choose to operate at a lower priority until it falls behind its transfer rate goals.

Video BackEnd

The Video Backend (Elmer) is also the video timing generator. It contains 8 interrupts which can be programmed to occur at the beginning of any video line. The interrupt line is connected to the I/O Chip (Mazda). The I/O chip contains information as to how the interrupts are vectored. These

interrupts are crucial to obtaining tear free updates. Their use is detailed in the **High Level Software** section.

The Video Backend (Elmer) provides a register which can be read by the CPU to determine the current vertical position (Y coordinate) of the monitors raster beam. This register is used in conjunction with the programmable interrupts for tear free updates.

Error Handling

The E&WC implementation of Wilson includes several Channels which can transfer data. These channels may encounter an error when either reading data or writing data. They may also encounter Retry indications. Retries are handled automatically by the Bus Interfaces. Errors must be handled differently.

If an E&WC Channel encounters a read error it will stop the channel and interrupt the CPU. The Channel which encountered the error will have its error indicator set. An E&WC Channel will never receive a error on a write (main memory will not generate an error & BLT indicates its errors directly to Mazda). Main memory also guarantees that it will not alias any addresses

Here is a summary of the potential errors and the action taken:

- XJS performs a write to a device on BLT which doesn't exist. BLT generates an **Error*** signal to Mazda. Mazda interrupts an XJS.
- XJS performs a write to a device on BLT which generates an error. BLT returns a **Delayed Error*** signal to Mazda. Mazda interrupts an XJS.
- XJS performs a read to a device on BLT which doesn't exist. BLT returns a Read Error Packet. E&WC returns a **SlvTERR*** to the Memory Controller. Memory Controller returns a **TERR*** to the requesting CPU.
- XJS performs a read to a device on BLT which generates an error. Same as above.
- BLT performs a write to an invalid address. This cannot occur. No addresses in the Motherboard's address space will generate an error for a write operation.
- BLT performs a read to an invalid address. E&WC passes the read transaction to the Memory Controller. The Memory Controller returns a **TERR*** to the E&WC. The E&WC returns a Read Error packet to BLT.
- E&WC performs a write to an invalid local memory address. This cannot occur.
- E&WC performs a write to BLT which generates an error. Mazda receives a **Delayed Error*** signal from BLT.

- E&WC performs a read from main memory which generates a parity error. Memory Controller returns a **SlvTERR*** to the E&WC. The E&WC halts the channel and interrupts an XJS.

- E&WC performs a read from BLT which generates an error. BLT returns a Read Error Packet. E&WC halts the channel and interrupts a CPU.

Power Consumption

Rough estimates for 50 MHz operation:

38 outputs operating at 8 MHz with 50 pF load	0.38
25 outputs operating at 12.5 MHz with 20 pF load	0.16
136 outputs operating at 12.5 MHZ with 20 pF load	0.85
Internal -- guesstimate	1.00
TOTAL	2.38 Watts

Gate Count

Currently estimated at 60000 used gates.

Reset

Reset clears all active transactions in both Bus Interfaces. Reset clears all **Active** bits in the the Channel registers and halts all channel operations. Reset clears all data buffers and associated pointers.

Reset does not clear any state registers which are visible to the CPU except for the **Active** bits.

Interrupts

Each RectRegion Destination Resource Channel, RectRegion Source Resource Channel, and Sequencer Channel can be programmed to interrupt either of the 2 XJS CPUs or other devices located on the Expansion I/O. The Interrupt concept is also used internally to synchronize the Sequencer Channels with the RectRegion Source Resource and the RectRegion Destination Resource.

The RectRegion Destination Resource and the RectRegion Source Resource can generate interrupts at page boundaries and at the completion of a transfer. These interrupts can be used to restart a Sequencer channel. The Sequencer Channel can then reload the the Resource's registers and restart it. The Sequencer Channel enters a dormant state and waits for another interrupt after the Resource Channel has been restarted.

The Expansion I/O of Jaguar (BLT) defines two interrupt architectures. The first architecture defines an interrupt line which is connected to Mazda. Mazda can either reflect the interrupt directly to one of the CPU's or translate it into a memory mapped write transaction. The second architecture is based on memory mapped writes. A BLT node can only receive interrupts through a memory mapped write. Each node has a specified space (highest possible 32 byte memory addresses) which are defined to be the interrupt space. If the card desires to receive interrupts it should decode writes to these locations.

The Jag1 motherboard implements the memory mapped architecture for interrupts in addition to its own internal scheme. The E&WC will decode writes to the address space from `$8fff ffc0` to `$8fff ffff` as interrupts. The E&WC knows about 10 potential interrupt destinations. These include the 2 XJS CPU's and the 8 Sequencer Channels. The interrupts are allocated as follows:

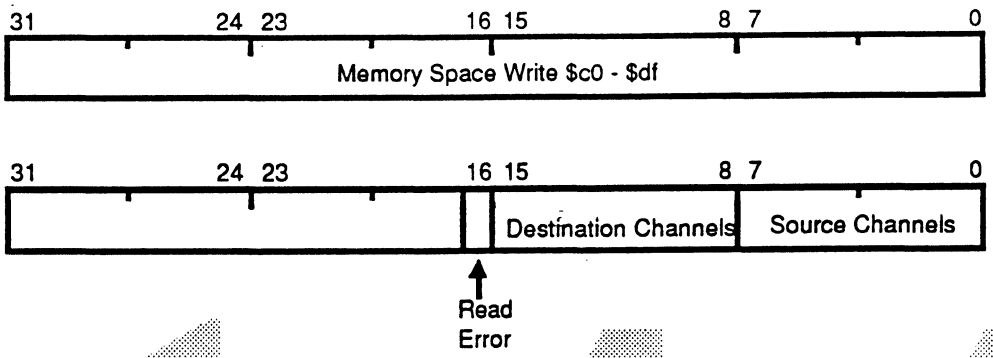
Address Range		Interrupt Destination
$8fff ffc0	$8fff ffdf	Processors
$8fff ffe0	$8fff ffe7	Sequencer Channels 0-7
$8fff ffe8	$8fff ffff	Reserved

If a read error is received by a Channel an interrupt is generated to Processor 0. The channel's normal interrupt will not be generated.

The following sources can interrupt the CPUs:

- Write to memory location in the range $8fff ffc0 to $8fff ffdf (32 possible sources).
- RectRegion Source Resource Channels (8)
- RectRegion Destination Resource Channels (8)
- A Read Error generated by a Sequencer Channel or a Source Resource Channel

Two interrupt status registers are available for the CPU to determine the source of the interrupt. These registers contain bits which are set when the interrupt occurs. They are defined below.



Interrupt Source Registers

Each Interrupt Source Register has two corresponding Interrupt Mask Registers. Their is one mask bit for each of the corresponding system CPU's. The CPU's will only receive interrupts which have their corresponding bit set to one. When a CPU reads the Interrupt Source Registers the bits in the register are automatically cleared. Only the bits which have the corresponding bits in the Mask register set are cleared. The other bits (if any) are left set for the other CPU. This requires two different address locations for the Interrupt Source Registers and the CPU must know which one to read.

The two Interrupt Lines from the E&WC are open collecter or'ed with the outputs of Mazda. Their is one line for each CPU.

The interrupt structure for E&WC is not finalized.

Power Down (Sleep Mode)

The E&WC will enter a sleep mode when the Reset line is pulled low. Power consumption will be minimized but internal state will be maintained. All current transactions will be lost. Software must guarantee that all channels are in the inactive state before entering sleep mode.

Memory Map

The Wilson Hardware is assigned the address range of `$8800 0000` to `$8fff ffff`. The Expansion I/O is assigned the address range `$9000 0000` to `$ffff ffff`. Each slot (node) is assigned 256 megabytes for a maximum of 7 nodes. The nodes are numbered 9 through F. Node 9 is assigned address space `$9000 0000` to `$9fff ffff`. Wilson decodes all local transactions on the XJS Bus which are in the Wilson Address Space or in the Expansion I/O Space. All transactions in the Wilson Address Space are responded to by the Wilson Hardware. All transactions

in the Expansion I/O Space are passed to BLT by the E&WC. The E&WC uses the 4 most significant bits of address to obtain the node number for BLT.

All transactions by expansion cards to Nodes 0 through 8 are passed by BLT to the E&WC. The E&WC responds to some of the transactions and passes others to the XJS Address Bus. It responds directly to addresses in the $8800 0000 to $8fff ffff. It passes addresses in the range $0000 0000 to $87ff ffff to the XJS Address Bus.

Node Identification Address: The address $8c00 0000 is defined as the Node Identification Address. If the E&WC receives a read from BLT in this address space it will return the node number of the requester as the data. This only occurs when the request is made from BLT.

Memory Map

$8800 0100	$8800 013f	RectRegion Destination Resource Ports 0-7
$8800 0200	$8800 023f	Pixel Munger Ports 0-7
$8800 0300	$8800 031f	Blender Ports 0-3
$8800 8000	$8800 81ff	Wilson RectRegion Source Resource Channels 0-7
$8800 9000	$8800 91ff	Wilson RectRegion Destination Resource Channels 0-7
$8800 a000	$8800 a1ff	Pixel Munger Channels 0-4
$8800 b000	$8800 b1ff	Blender Channels 0-1
$8800 c000	$8800 c1ff	Sequencer Channels 0-7
$8800 d000	$8800 d007	Interrupt source registers 0-1
$8800 d008	$8800 d017	Interrupt Mask Registers
$8c00 0000	$8c00 0000	Node indentification address
$8fff ffc0	$8fff ffff	Interrupts

all ranges not listed above are undefined at this time.

Section 6

Programmers Model

Programmer's Model

The hardware programmer's model is described in this section primarily with examples. There are several ways the Wilson Hardware in the E&WC can be utilized. The method used depends on the type of operation being performed.

A queuing model is utilized for synchronous operations. These operations might be requested by an application (for example move a block of data from location a to location b and return when done).

The second type of operation which appears important is a repetitive. A repetitive operation would be used for decompressed video input or live video input (for example a video input card sends a stream to the motherboard's frame buffer).

Virtual Memory

An application requests that a region be moved from on location to another. This is translated into a request to the Wilson Manager. The Wilson Manager allocates a RectRegion Source Resource Channel (RRSRC1) and a RectRegion Destination Resource Channel (RRDRC1) for the operation. It uses a pre-allocated Sequencer Channel (SC1). Assume the source is in virtual memory (system memory) and is fragmented. Assume the destination is the frame buffer (basically contiguous). The following commands are placed in main memory by the CPU.

```
Start:    MoveI   RRSRC1.SAP      #$0003 4000      //first page address
          MoveI   RRSRC1.SA       #$0003 4000      //first page address
          MoveI   RRSRC1.STRWID   #$0040 0036      //Width=36 & Stride=40
          MoveI   RRSRC1.WIDC     #$0000 0036      //WidthCount=36
          MoveI   RRSRC1.RAT      #$0000 0000      //rate=as fast as possible
          MoveI   RRSRC1.LEN      #$0000 0300      //we will move 300 bytes
          MoveI   RRSRC1.DA       #$0444 0394      //destination address of stream
          MoveI   RRSRC1.CTL      #$0100 0406      //set control bits
                                                   //direction=0,interrupt length=1
                                                   //interrupt page=1,priority=0
                                                   //stream=0,global=0,pixelsize=4bytes
                                                   //constant=no
                                                   //interrupt vector points to SC1
Middle:   MoveI   RRSRC1.E        #$0000 0000      //clear error register
          MoveI   RRDRC1.DAP      #$4000 4400      //destination address
```

```
MoveI   RRDRC1.DA       #$4000 4400     //destination address
MoveI   RRDRC1.STRWID   #$0080 0036     //width=36, stride=80
MoveI   RRDRC1.WIDC     #$0000 0036     //
MoveI   RRDRC1.LEN      #$0000 0300     //length isn't really needed.
MoveI   RRDRC1.CTL      #$0000 0000     //set up destination ctl register.
MoveI   RRDRC1.A        #$ffff ffff     //Start Destination Channel
MoveI   RRSRC1.A        #$ffff ffff     //Start Source Channel
Wait                                    //wait for next page cross
MoveI   RRSRC1.SA  #$0004 8000          //second page address
MoveI   RRSRC1.A   #1
Wait
MoveI   RRSRC1.SA  #$0032 5000
MoveI   RRSRC1.A   #1
Wait
Halt
```

The CPU then loads the SC1.IP register with the value of "Start:" . This starts the sequencer executing instructions at "Start:" The sequencer loads all the registers associated with the RRSRC1. It loads all the registers associated with RRDRC1. It then starts both the source and destination by changing their **Active** bits to ones. The SC1 encounters the wait instruction and stops execution. It will not continue execution until it receives an interrupt. The RRSRC1 reads the data from memory and sends it to the RRDRC1. RRDRC1 writes the data to the framebuffer. When RRSRC1 crosses a page boundary it stops execution and interrupts the SC1 (its **Interrupt on Page Boundary=1** and its **Interrupt Vector** points to SC1). The SC1 resumes executing instructions. It loads a new page address into the RRSRC1.SA register. It then restarts the Source Resource by loading its **Active** bit with a one. The RRSRC1 reads more data from memory and sends it to RRDRC1 which writes it into the framebuffer. The **Length** register of the RRSRC1 eventually reaches zero and the RRSRC1 stops execution and interrupts SC1. SC1 executes the halt instruction (it could interrupt the processor at this point to indicate completion).]

Queuing Model

A Wilson Sequencer Channel can be set up as an execution queue. Operations which must be performed are placed into the queue. The Wilson Sequencer Channel will execute them in order and report their completion. An example will help clarify the operation.

If we label the preceding example as "Transfer #1" we can demonstrate the queuing model. Let us assume we have another transfer similar to Transfer #1 which is called Transfer #2. If we replace the halt instruction (the last instruction) in Transfer #1 with all the commands needed to perform Transfer #2 the Sequencer will continue and execute Transfer #2 after it has completed Transfer #1. A better approach would be to place an MoveI instruction in between the two transfers to generate an interrupt to the CPU to indicate that Transfer #1 is complete. This would look like:

```
Start1:  MoveI   RRSRC1.SAP     #$0003 4000     //first page address
         .....
         MoveI   RRDRC1.CTL     #$0000 0000     //set up destination ctl register.
         MoveI   RRDRC1.A       #$ffff ffff     //Start Destination Channel
         MoveI   RRSRC1.A       #$ffff ffff     //Start Source Channel
         Wait                                   //wait for next page cross
         MoveI   RRSRC1.SA  #$0004 8000         //second page address
         MoveI   RRSRC1.A   #1
         Wait
         MoveI   RRSRC1.SA  #$0032 5000
```

```
          MoveI   RRSRC1.A    #1
          Wait
          MoveI   INTERRUPT  #$0000 0000          //generate interrupt
Start2:   MoveI   RRSRC1.SAP     #$0003 4000      //first page address
          ....
          halt
```

This work fine if the channel has not been started. But how do you add the second transfer to the queue once the Sequencer has already been started? The next section explains.

Dynamic Queue Extension

Let's assume that the instruction sequence for SC1 has been built and that SC1 is executing its instruction sequence. It is currently located "Middle:" The CPU receives a request to perform Transfer #2. It desires to place Transfer #2 at the end of the execution queue of SC1.

```
Start:    MoveI   RRSRC1.SAP     #$0003 4000      //first page address
          ....
          MoveI   RRSRC1.DA      #$0444 0394      //destination address of stream
          MoveI   RRSRC1.CTL     #$0100 0406      //set control bits
                                                  //interrupt vector points to SC1
Middle:   MoveI   RRSRC1.E       #$0000 0000      //clear error register
          MoveI   RRDRC1.DAP     #$4000 4400      //destination address
          ....
          MoveI   RRSRC1.A    #1
          Wait
          MoveI   RRSRC1.SA   #$0032 5000
          MoveI   RRSRC1.A    #1
          Wait
          Halt
```

The CPU places the instruction sequence for Transfer #2 at the end of the SC1 execution queue (after the Halt instruction). It leaves the halt instruction intact until the complete instruction sequence for Transfer #2 is in place. The CPU then replaces the Halt Instruction with either the first instruction of Transfer #2 or a null operation instruction. The CPU must then perform a write to the SC1.IP register of the address of the old Halt instruction.

This dynamic queue extension is guaranteed to work for both the case of the SC1 having already completed Transfer #1 and the case of SC1 still executing Transfer #1. If Transfer #1 is still executing the write to SC1.IP is ignored because the Sequencer Channel is active. Eventually the Sequencer Channel reaches Transfer #2 and executes it. If Transfer #1 has completed and the halt instruction has been executed the write to SC1.IP restarts the Sequencer Channel.

Stopping a Channel

A Sequencer Channel can be stopped by writing a zero into its **Active** bit. This will cause it to complete its current instruction and then stop. It will not, however, cause any Source Resource Channels which the sequencer has started to stop their transfers of data.

Source Resources can be stopped by writing a zero into their **Active** bit. This may or may not have catastrophic side effects. The answer is currently unknown and will be determined later during the implementation.

Section 7

Examples

This provides examples of how the software and hardware work together to perform the functions desired.

Example Wilson Data Flows

This section provides some of the data flows which are thought to be useful.

1. Live Video Window directly to Frame Buffer or Memory with Clip Alpha. A video device located on the Expansion I/O sends a stream of pixel data to the motherboard's RectRegion Destination Resource. The video device has a full 1 bit clip alpha channel which it expands to 8 bits when sending the data. The resulting stream is in the `k32_RGB` format. The RectRegion Destination Resource receives the stream from the video device and writes all pixels which have an alpha value greater than zero to memory (frame buffer). The region shape is determined by the rectangular region described in the RectRegion Destination Resource Channel and by the clip alpha data. The clip alpha data provides for arbitrary shapes. The pixels are translated to `k16_GRAY` from `k32_RGB` by the RectRegion Destination Resource if the framebuffer is 8 bits per pixel.

2. Live Video Window directly to Frame Buffer or Memory without Clip Alpha. Same as above except the video device does not contain a full 1 bit clip alpha channel. The alpha data sent with the stream is invalid (unknown value). The E&WC on the motherboard receives the data and routes it to the Pixel Munger. The RectRegion Source Resource reads the Clip Alpha information from main memory, expands it to a full 8 bits, and sends it to the Pixel Munger. The Pixel Munger combines the alpha data with the Pixel data received from the video device. The Pixel Munger sends the data to the RectRegion Destination Resource. The RectRegion Destination Resource writes all data which has an alpha!=0 to memory.

3. Live Video Window directly to Frame Buffer with text or Graphics Overlay. Same as above (the clip alpha prevents writing over the text and graphics)

4. Live Video Window directly to Frame Buffer with Clip Alpha and Antialiased Text Overlay. A video device located on the Expansion I/O sends a stream of pixel data to the motherboard. The video device has a full 1 bit clip alpha channel which it expands to 8 bits when sending data. The E&WC receives the stream from the video device and routes it to the Blender. A full premultiplied overlay image with associated

alpha channel exists in main memory. The memory image is the same size as the rectangular region which is being sent by the video device. A RectRegion Source Resource fetches the image from main memory and creates a stream which it routes to the Blender. The Blender receives the streams from the video device and the RectRegion Source Resource. It performs the following operation:

```
RGB(result)   = RGB(memory) + (1-ALPHA(memory))  *  RGB(video)
ALPHA(result) = ALPHA(memory) + (1-ALPHA(memory)) * ALPHA(video)
```

Alpha override is enabled such that if the ALPHA of the video is zero the resulting alpha is also zero otherwise the alpha is properly calculated. The data is routed from the Blender to the RectRegion Destination Resource where all pixels which have an alpha value greater than zero are written to memory (or frame buffer).

5. Live Video Window directly to Frame Buffer or Memory with Antialiased Text Overlay without Clip Alpha. Same as above except the video device does not contain a full 1 bit clip alpha. The alpha data sent with the video stream is invalid (unknown value). The E&WC on the motherboard receives the data and routes it to the Pixel Munger. The RectRegion Source Resource reads the Clip Alpha information from main memory, expands it to 8 bits, and sends it to the Pixel Munger. The Pixel Munger combines it with the Pixel data received from the video device. The Pixel Munger sends the data to the Blender. A full premultiplied overlay image with associated alpha channel exists in main memory. The memory image is the same size as the rectangular region which is being sent by the video device. The RectRegion Source Resource fetches the image from main memory and creates a stream which it routes to the Blender. The Blender receives the streams from the Pixel Munger and the RectRegion Source Resource. It performs the following operation:

```
RGB(result)   = RGB(memory) + (1-ALPHA(memory))  *  RGB(video)
ALPHA(result) = ALPHA(memory) + (1-ALPHA(memory)) * ALPHA(video)
```

Alpha override is enabled such that if the ALPHA of the video is zero the resulting alpha is also zero otherwise the alpha is properly calculated. The data is routed from the Blender to the RectRegion Destination Resource where all pixels which have an alpha value greater than zero are written to memory (or frame buffer).

6. Full Front to Back Compositing with Live Video. (this probably won't work at 60 frames a second because of bandwidth limits). A Video devices sends the live video stream with a full 8 bit alpha channel to the E&WC. The E&WC routes the stream to the Blender. The Blender performs a premultiply operation on the live video and sends the video back to the input of the Blender. The RectRegion Source Resource fetches the image which is located behind the live video. This image contains an alpha channel which is all ones (255). It is the same size as the live video image and is basically the composite of all things behind the video. The RectRegion Source Resource routes this data to the blender. The blender composites the two images and sends the result back into the blender. The RectRegion Source Resource fetches the image which is located in front of the video. This image contains a true alpha channel. It is the same size as the live video image and is basically the composite of all images in front of the video. The RectRegion Source Resource routes this data into the blender. The Blender composites these two inputs and sends the result to the RectRegion Destination Resource. The RectRegion Destination Resource writes

the data to the frame buffer (memory). Note that no clipping is performed in this example.

7. Memory, Frame Buffer, or Back Buffer Rectangular Clear Operation. The CPU desires to clear a rectangular section of memory. The CPU sets up a RectRegion Source Resource with a constant value. The RectRegion Resource sends the value repeatedly to the RectRegion Destination Resource. The RectRegion Destination Resource writes the values into memory.

8. Memory, Frame Buffer, or Back Buffer Non-Rectangular Clear Operation. The CPU desires to clear a a section of memory. The CPU sets up a RectRegion Source Resource with a constant value. The RectRegion Source Resource sends this value repeatedly to the Pixel Munger. The CPU sets up a second RectRegion Source Resource to fetch from memory a one bit clip alpha stream, expand it to 8 bits, and send it to the Pixel Munger. The Pixel Munger the alpha channel and the pixel intensity channel to generate a 32_RGB output stream. It sends the output stream to the RectRegion Destination Resource. The RectRegion Destination Resource writes all pixels for which alpha!=0 into memory.

9. Standard Rectangular Blit Operation. The CPU desires to transfer an off screen rectangular region to the framebuffer. The RectRegion Source Resource reads the data from the source area of memory and creates a stream of data. The stream is transferred to the RectRegion Destination Resource. The RectRegion Destination Resource writes the rectangular region to memory.

10. Standard Non-rectangular Blit Operation. The CPU desires to transfer an off screen non-rectangular region to the framebuffer. The RectRegion Source Resource reads the pixel data (with or without alpha) of a rectangular region from the source memory. The RectRegion Source Resource reads a 1 bit alpha (mask) data from another area of memory and expands it to 8 bits. The two channels are sent to the Pixel Munger. The Pixel Munger replaces the alpha on the pixel data with 0 if the mask data is zero (pixel should not be written). If the mask data is not zero the Pixel Munger does not change the alpha value of the pixel stream. The results of the Pixel Munger are sent to the RectRegion Destination Resource where the pixels are written to the framebuffer if alpha!=0.

11. Creating a premultiplied image. The CPU desires to create a premultiplied image from a standard image. The RectRegion Source Resource reads the pixel data from memory and routes it to the Blender. The Blender performs a premultiply operation and sends the result to the RectRegion Destination Resource. The RectRegion Destination Resource writes the data back into memory.

12. Compositing two premultiplied images. The CPU desires to composite two back buffers into a single backbuffer. The RectRegion Source Resources reads the two input pixel maps from the source memory and sends them to the Blender. The Blender performs the following operation:

```
RGB(result)   = RGB(memory1) + (1-ALPHA(memory1))   *  RGB(memory2)
ALPHA(result) = ALPHA(memory1) + (1-ALPHA(memory1)) * ALPHA(memory2)
```

The Blender routes the data to the RectRegion Destination Resource. The RectRegion Destination Resource writes the result into memory.

Inexpensive (Low Quality) Live Video Example

Scenario: An inexpensive video card is located on the Expansion I/O. It contains at most two line buffers worth of storage (for interpolation & data storage). It contains a 1 bit buffer the size of an NTSC screen which contains "mask" data (clip alpha). The mask data is set up by the CPU and the video card expands each single bit into a full byte of alpha using bit replication. All data transmitted by the video device is 32_RGB format.

This system is labeled as low quality because it is incapable of avoiding frame tears (it lacks a full backbuffer). The video data is sent directly to the Frame Buffer which is mapped in a physically contiguous block. The instantaneous rate of the NTSC output will be (4 bytes) * (12.27MHz *1.01) = 49.57 MB/sec. The rate will be doubled (99.14 MB/sec) if interpolation is performed. A line buffer is available therefore the instantaneous rate can be averaged over the horizontal retrace time to obtain (640/780)*99.14=81.35 MB/sec. If the data sent into the framebuffer is twisted it will require only (3/4) of this bandwidth or 61 MB/sec. If the displayed window is not the full width (640 pixels) the necessary bandwidths will be lower and can be calculated from the width ratio.

When the system decides it is appropriate to place a live video window on the screen the following sequence of events occurs:

- Application desires live video displayed on the screen.
- Application specifies the desired characteristics to the Video Toolbox (size, location, TGraphPort, quality, frequency, etc...). The application may specify back off points. The application may request synchronization information. The application may specify how it would like to be informed if performance degradation must occur. The application must specify how a failure is handled.
- Video Toolbox determines the details of the operation the application desires. It may inform the application that the operation is not possible (based on hardware information it has, no video input or already busy, for example). Video Toolbox determines the exact data movement operations needed.
- Video Toolbox may add its own back off points, synchronization information requests, degradation information requests, and failure methods.
- Video Toolbox, View System, Layer Manager, and Window Manager work to determine the visible region of the video. The clip alpha data in the video device is updated/created. This can be accomplished in one of four methods based on the card's implementation:1) CPU draws directly into Mask space using 1 bit per pixel drawing routines, 2) CPU draws directly into Mask space using 8 bits per pixel

drawing routines, 3) the CPU must call card specific ROM Routines to change the mask data, or 4) the CPU draws by sending a stream to the mask region. A PixRectRegion is created to describe the rectalinear region which must be transferred from the video device.

- Video Toolbox creates a TVideoStream and TMemoryRectaStreamDestination objects to describe the data flow needed. Video Toolbox creates a TContiuousSchedule object to further describe the transfer. The TSchedule object may contain references to TReply(), TConstraints(), and TDegrade() objects.

- Video Toolbox executes an Allocate() call on the TSchedule() object.

- Wilson Manager determines the channels needed from the references and pointers contained in the TSchedule() object. A VideoSource Resource Channel on the video device is needed and a RectRegion Destination Resource Channel is needed. If the channels cannot be allocated or software multiplexed either a back off point is attempted or a failure is returned to the Video Toolbox.

- Wilson Manager determines the general bandwidth needed for the operation. It uses the size of the transfer, the frequency, the bits per pixel information, and the general operation to determine the overall bandwidth. It requests the bandwidth from the BWM. If the BWM cannot allocated the bandwidth the Wilson Manager may request a back off point. The Wilson Manager calls the BWM and requests Continuous Bandwidth on BLT, the E&WC BLT connection, the E&WC memory connection, and the Frame Buffer. The bandwidth allocated is the maximum peak transfer rate of the video input device. If the bandwidth is not available video cannot be displayed. The bandwidth allocated for an NTSC window 640 pixels wide must be at least 81.35 MB/sec.

- Wilson Manager returns the status of the operation to the Video Toolbox via the indicated Reply method.

- Application or Toolbox performs an Activate() on the TSchedule Object.

- Wilson Manager performs device driver calls to set up the registers in the Wilson Hardware for the appropriate devices. The Wilson Manager performs virtual to physical address translation. The VideoIn() device driver is called and its registers are set up with the appropriate values to describe the transfer. The Wilson MotherBoard driver is called and the registers of the RectRegion Destination Resource Channel are set up to describe the destination window.

- Wilson Manager registers the transfer with the MBWM along with any of the constraints which are important to the MBWM. When the bandwidth is available for the transfer the MBWM starts the transfer.

- VideoIn Source Resource Channel waits for the appropriate data to be received on its input half. When the data is available it creates a stream of data which it sends to the motherboard's RectRegion Destination Resource Channel. The clip alpha data is attached to the stream by the video device.

- RectRegion Destination Resource Channel receives the stream and writes the data to the appropriate area of the framebuffer.

- RectRegion Destination Resource Channel or the video device generates an interrupt at the completion of the transfer.

- MBWM receives an interrupt when the transfer is completed. The MBWM indicates the transfer has completed to the Wilson Manager. If the MBWM is unable to complete the transfer in the specified amount of time or within other constraints it informs the Wilson Manager.

- Wilson Manager performs any Reply Methods or other Constraint checks (synchronization).

- Wilson Manager performs device driver calls to reestablish the hardware registers. Wilson Manager reregisters the transfer with the MBWM. This sequence is repeated until the TSchedule object is de-activated.

- Video Toolbox calls the Deactivate() procedure of the TSchedule object to stop the video.

- Video Toolbox calls the DeAllocate() procedure of the TSchedule object to deallocate the bandwidth and the Wilson Channels.

Quality Live Video Support Example

Scenario: A video card is located on the Expansion I/O. It contains a full backbuffer for the incoming video. The system is capable of full frame rate conversion and tear avoidance. It contains a 1 bit buffer the size of an NTSC screen which contains "mask" data (clip alpha). The mask data is set up by the CPU and the video card expands each single bit into a full byte of alpha using bit replication. All data transmitted by the video device is 32_RGB format. The video input card also contains one or more programmable interrupts. The interrupts can be programmed to occur at a particular point during the input frame. The interrupts are controlled and received by the MBWM.

When the system decides it is appropriate to place a live video window on the screen the following sequence of events occurs:

- • • • • • • • first 8 steps same as above...

- Wilson Manager determines the general bandwidth needed for the operation. It uses the size of the transfer, the frequency, the bits per pixel information, and the general operation to determine the overall bandwidth. It requests the bandwidth from the BWM. If the BWM cannot allocated the bandwidth the Wilson Manager may request a back off point. The Wilson Manager calls the BWM and requests Repetitive Burst Bandwidth on BLT, the Wilson BLT connection, the Wilson memory connection, and the Frame Buffer. The Wilson Manager determines the maximum burst bandwidth which can be sustained between the source and destination of the video. This is calculated from the maximum rates sustainable by each device. The Wilson

Manager selects a burst bandwidth for the transfer which is less than the maximum burst rate and less than the available bandwidth for each device. A value is selected which makes the window of valid constraints for the tear free update a reasonable size.

- Wilson Manager returns the status of the operation to the Video Toolbox via the indicated Reply method.

- Application or Toolbox performs an Activate() on the TSchedule Object.

- Wilson Manager performs device driver calls to set up the registers in the Wilson Hardware for the appropriate devices. The Wilson Manager performs virtual to physical address translation. The VideoIn() device driver is called and its registers are set up with the appropriate values to describe the transfer. The Wilson MotherBoard driver is called and the registers of the RectRegion Destination Channel are set up to describe the destination window.

- Wilson Manager registers the transfer with the MBWM along with any of the constraints which are important to the MBWM.

- MBWM sets up any interrupts it needs for handling the constraints (tear free updates). When the bandwidth is available for the transfer and the constraints are met the MBWM starts the transfer.

- VideoIn Source Resource Channel creates a stream of data which it sends to the motherboard's RectRegion Destination Resource Channel. The clip alpha data is attached to the stream by the video device.

- RectRegion Destination Resource Channel receives the stream and writes the data to the appropriate area of the framebuffer.

- RectRegion Destination Resource Channel generates an interrupt at the completion of the transfer.

- MBWM receives an interrupt when the transfer is completed. The MBWM indicates the transfer has completed to the Wilson Manager. If the MBWM is unable to complete the transfer in the specified amount of time or within other constraints it informs the Wilson Manager.

- Wilson Manager performs any Reply Methods or other Constraint checks (synchronization) which were requested.

- Wilson Manager performs device driver calls to reestablish the hardware registers. Wilson Manager reregisters the transfer with the MBWM. This sequence is repeated until the TSchedule object is de-activated.

- Video Toolbox calls the Deactivate() procedure of the TSchedule object to stop the video.

- Video Toolbox calls the DeAllocate() procedure of the TSchedule object to deallocate the bandwidth and the Wilson Channels.

Single Back Buffered Animation

Animation is generated by the CPU in main memory. Wilson transfers the completed frame from the backbuffer to the frame buffer avoiding tears. Note that the backbuffer is located in virtual memory and is not necessarily contiguous in physical memory.

- Application desires perform animation.

- Application specifies the desired characteristics to the Animation Toolbox (size, location, TGraphPort, quality, frequency, CPU cycles needed per frame, etc...). The application may specify back off points. The application may request synchronization information. The application may specify how it would like to be informed if performance degradation must occur. The application must specify how a failure is handled. Some (or all) of these characteristics may actually be determined by the animation toolbox.

- Animation Toolbox determines the details of the operation the application desires. Animation Toolbox determines the exact data movement operations needed and the CPU cycles

- Animation Toolbox may add its own back off points, synchronization information requests, degradation information requests, and failure methods.

- Animation Toolbox requests the CPU cycles for the animation from the CPU Cycle Manager. If the cycles cannot be obtained a compromise or back off position must be negotiated. The Scheduler guarantees the application/toolbox will receive the proper number of cycles (when its not "blocked"). If the scheduler fails it will inform the application/toolbox/Wilson Manager.

- Animation Toolbox, View System, Layer Manager, and Window Manager work to determine the visible region of the video. A clip alpha description is created in main memory A PixRectRegion is created to describe the rectalinear region which must be transferred from the back buffer to the screen.

Video Toolbox creates a TVideoStream and TMemoryRectaStreamDestination objects to describe the data flow needed. Video Toolbox creates a TContiuousSchedule object to further describe the transfer. The TSchedule object may contain references to TReply(), TConstraints(), and TDegrade() objects.

- Animation Toolbox creates TStream objects to describe the data flow needed. These are a TMemoryRectaStrippedPixelStream() object (for reading the back buffer), a

TMemoryRectaAlphaStream() object (for reading the clip alpha), a TPixelMungerStream Object (for putting alpha and back buffer together), and a TMemoryRectaStreamDestination() object (for writing the data to the screen). Animation Toolbox creates a TRepetSchedule object to further describe the transfer. The TSchedule object may contain references to TReply(), TConstraints(), and TDegrade() objects. The TSchedule object indicates frame by frame synchronization is needed.

- Animation Toolbox executes an Allocate() call on the TSchedule() object.
- Wilson Manager determines the channels needed from the references and pointers contained in the TSchedule() object. Two RectRegion Source Resource Channels, two Sequencer channels, a RectRegion Destination Resource Channel, and a Pixel Munger Channel are needed. If the channels cannot be allocated or software multiplexed either a back off point is attempted or a failure is returned to the Animation Toolbox.
- Wilson Manager requests the Kernel lock the necessary pages in memory.
- Wilson Manager determines the general bandwidth needed for the operation. It uses the size of the transfer, the frequency, the bits per pixel information, and the general operation to determine the overall bandwidth. It requests the bandwidth from the BWM. If the BWM cannot allocated the bandwidth the Wilson Manager may request a back off point. The Wilson Manager requests Repetitive Burst Bandwidth on the E&WC memory connection, main memory, and the Frame Buffer. The Wilson Manager determines the maximum burst bandwidth which can be sustained between the source and destination. This is calculated from the maximum rates sustainable by each device. The Wilson Manager selects a Burst bandwidth for the transfer which is less than the maximum burst rate and less than the available bandwidth for each device. A value is selected which makes the window of valid constraints for tear free updates an acceptable size.
- Wilson Manager returns the status of the operation to the Animation Toolbox via the indicated Reply method.
- Application/Toolbox render first off screen frame of animation.
- Application or Toolbox performs an Activate() on the TSchedule Object.
- Wilson Manager performs device driver calls to set up the registers in the Wilson Hardware for the appropriate devices. The Wilson Manager performs virtual to physical address translation. The Wilson Manager calls the motherboard's Wilson Driver to set up two instruction sequences in main memory for the two Sequencer Channels. The sequence channels work in conjunction with the RectRegion Source Resource Channels to perform gather operations. The instruction sequences look as follows:

```
MoveI   RRSRC1.SAP     #$0003 4000        //first page address
MoveI   RRSRC1.A       #1                 // restart Win Read Chan
Wait                                      // wait for next page cross
MoveI   RRSRC1.SAP     #$0033 4000        //second page address
```

```
MoveI   RRSRC1.A        #1                  // restart Win Read Chan
Wait
MoveI   RRSRC1.SAP      #$0001 4000         //third page address
MoveI   RRSRC1.A        #1                  // restart Win Read Chan
Wait
Halt
```

The Wilson MotherBoard driver is called and the registers of the RectRegion Destination Resource Channel and the Pixel Munger Channel are set up to describe the destination window and the incoming stream formats.

- Wilson Manager registers the transfer with the MBWM along with any of the constraints which are important to the MBWM.

- MBWM sets up any interrupts it needs for handling the constraints (tear free updates). When the bandwidth is available for the transfer and the constraints are met the MBWM starts the transfer.

- Both RectRegion Source Resource Channels create streams of data which they send to the motherboard's Wilson Pixel Munger. The first stream is the pixel data. The second stream contains the clip alpha information.

- The Pixel Munger attaches the clip alpha to the Pixel data and send the stream to the RectRegion Destination Resource Channel.

- RectRegion Destination Resource Channel receives the stream and writes the data to the appropriate area of the framebuffer.

- RectRegion Source Resource Channel generates an interrupt to the Sequencer Channel when a page boundary is crossed. The Sequencer Channel loads the next page address and restarts the RectRegion Source Resource Channel. This operation is repeated for both RectRegion Source Resource Channels and Sequence Channels throughout the transfer.

- RectRegion Destination Resource Channel generates an interrupt at the completion of the transfer.

- MBWM receives an interrupt when the transfer is completed. The MBWM indicates the transfer has completed to the Wilson Manager. If the MBWM is unable to complete the transfer in the specified amount of time or within other constraints it informs the Wilson Manager.

- Wilson Manager performs any Reply Methods or other Constraint checks (synchronization) which were requested. It informs the toolbox/application that the transfer has been completed. Wilson Manager waits for a message from the application/toolbox.

- Toolbox/Application render the next frame of the sequence and send a message to the Wilson Manager.

- Wilson Manager performs device driver calls to reestablish the hardware registers. Wilson Manager reregisters the transfer with the MBWM. The Wilson Manager is able to reuse the Sequence Channel Instruction Sequences.

- The basic operations are repeated as long as the application/toolbox desire the animation.
- Animation Toolbox calls the Deactivate() procedure of the TSchedule object to stop the video.
- Animation Toolbox calls the DeAllocate() procedure of the TSchedule object to deallocate the bandwidth and the Wilson Channels.
- Wilson Manager unlocks the memory pages.

Recording & Displaying Sequential Frames

to be completed. compressed or uncompressed.

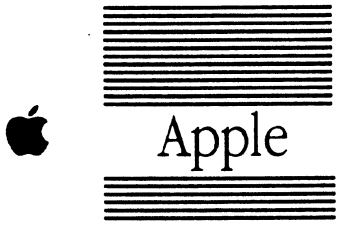
multiple destinations simultaneously.

Section 8

Issues

Issues

- The Wilson architecture and the hardware does not support frame buffers with CLUTs on their output. This is assumed acceptable.
- The cost and complexity of performing operations in forwards and reverse must be weighed against the benefits before a commitment is made.
- Multiple CPU transactions to BLT may be compressed into a single write transaction before being written out to BLT. Should this be attempted?
- Is the cost of the CLUT worth it?
- Does the software interface provide all the important functionality of "CopyBits"?
- Is it useful to write a constant instead of clipping at the destination?
- Do we want to support images which have not been pre-multiplied?
- Number of RectRegion Source Channels. Is eight enough?
- The priority mechanism is relatively simple. How much effort should be made in optimizing it? How much can be gained by optimizing it?
- Pins will be needed to support the decompression hardware. What will be needed?



BLT- System Expansion For Jaguar

External Reference Specification
Special Projects
Jaguar

November 14, 1989
Version 1.02

Brought to you courtesy of
Charles, Steve, Todd, and a host of others

Return Comments to:
Steve Roskowski
Phone: x4-4312
AppleLink: ROSKOWSKI1
MS: 60D

Contents

Section 1

Introduction

The reason an expansion system exists in a computer is to allow a user to add functionality to the basic system. For adding another Ethernet port, a synchronous serial line, or a GPIB card, Nubus is more than sufficient.

One of Jaguars claims is that it is a media machine. One quick look at the amount of bandwidth consumed by media transfers makes it obvious that Nubus will not cut it as a media expansion device. A single channel of NTSC video can consume greater then 60 megabytes per second of bandwidth[1]. This would be a little tough with the 38 megabytes per second that Nubus can deliver[2].

With this in mind, a new system was designed from the ground up. The result is BLT (bus like thing). It is in many ways a fundamental departure from all previous expansion systems. BLT is not a bus. It is a fully connected packet switched crossbar. It provides all of the facilities found in a conventional bus plus many new ones.

BLT is an extremely high performance interconnect. For normal reads it will out perform Nubus by a factor of two to four. For write operations it will be 5 to 10 times faster. Where BLT really shines is large block data transfers, exactly what is needed for media transfers. Here it will beat Nubus by a factor of 10 to 100. The raw bandwidth of BLT can exceed 400 megabytes per second between any two cards, and it allows several transactions to be conducted at the same time.

There are three fundamental goals for the Jaguar bus. First, it must supply sufficient bandwidth to enable a new graphics architecture. If the bus has enough bandwidth, graphics sources and destinations can be separated, which provides a clean,orthogonal, and efficient architecture. Sufficient is currently set at about 300 megabytes per second, for various obscure reasons.

The second goal is architectural expandability. BLT will be around for quite a while. It must be able to meet the demands of small computer busses for at least 10 years. The card interface must

---

[1]For those technically in the know, this number may seem high. It is approximately double the actual video data rate. The factor of two stems from a technique that avoids cross-field tears in the conversion from interlaced to non-interlaced display devices... the cost of quality.

[2]There are currently some proposals to increase Nubus performance by use of special burst transactions. This is a wonderful idea but still falls short of the needs of a media system.

provide all of the features needed today as well as providing a means of reaching the features we foresee in the future.

The final goal for the Jaguar bus is to make it easier to use than any previous bus. The Jaguar bus is unique in that it has a fully active backplane. This makes it possible to provide a great deal more functionality built into the interconnect than in a conventional backplane.

BLT is designed many levels of card interface. Some cards will use it as a simple 8 bit non-multiplexed bus. Others will use it as a 400 megabyte per second streaming interconnect. These diverse cards can communicate with each other without any difficulty and without either knowing anything about the other card.

An important point about BLT is not all the new features it offers but the old features that it improves on. It is multi-master, peer-to-peer, has strict software selectable priority levels with fairness within them, and supports full split transactions. Any other buzz words that can be applied to a bus probably can be applied to BLT.

This document is dedicated to a description of how BLT is used. It explains the card interface, the interrupt mechanism, firmware, and other items important to card designers. As such, this information is architectural to Jaguar. Once it becomes solidified it will be with the family for life. This implies that a great deal of care must be taken to specify the proper functionality for future growth. It is intended that BLT will fill the role of Nubus for Jaguar; the normal expansion system. Further it is a goal that Jaguar will not suffer from the current new bus phenomena that is plaguing the Macintosh, as every new machine has yet another new bus.

On the implementation level, Jag1 will implement a four node BLT. This provides three expansion slots.

The architecture is designed to expand to eight nodes, which provides seven expansion slots. A seven slot machine will require a fair amount of engineering over a 3 slot box; all the ASICs that make up BLT must be re-implemented.

On the opposite extreme, a one slot machine is also under consideration. This would remove all ASICs that make up BLT and use the Wilson interface chip as the "backplane".

All of these design centers have been considered and seem to be achievable.

Section 2

Concepts and Facilities

There are many new concepts introduced by BLT. Its basic architecture is very different from a standard bus. The bus interface is similar to todays busses, but there are some fundamental changes that make building a card much easier. BLT can be used as a simple bus. As such it provides all the features of any modern interconnect. However, BLT goes beyond any other bus by providing ultra-high performance, ease of use, and special operations, optimized for data transfers.

Table 1-1 is a list of the notably new and interesting features of BLT. It of course supports all the conventional bus transaction: reads and writes of all common sizes, and locks to guarantee atomic transactions in multi-processor environments.

Table 1-1
BLT feature list

400 Megabytes per second	Sufficient bandwidth to support multiple simultaneous live animation or video windows.
Arbitrary frequency synchronous card interface	Each card interfaces to BLT at its own frequency, optimized for on board timings
"Polymorphic" card interface	Available interface modes include 64 and 32 bit multiplexed address and data as well as 32, 16, and 8 bit separate address and data.
Multiple simultaneous flow controlled data streams	Multiple write "streams" can be transmitted from any card. Destinations can chose to block or receive any one at any time.
Built in write buffering	BLT contains several packets worth of write buffering per node. This allows write operations to complete in minimum time, significantly improving throughput.
Multiple levels of split transactions	A card can have multiple reads outstanding simultaneously. They can complete either in submission order or in response order depending on card design.

8 levels of strict priority	BLT supports up to 8 different priority levels for transaction. Absolute priority is maintained so high priority items are always ahead of low priority.
Live Insertion and Removal	Cards may be inserted or removed while the machine is operating.
High Performance Block Transfers	The natural operating mode of BLT is to transfer blocks of data. Single word transactions are special cases, a block of 4 bytes.

Important Terms

This document assumes a fair amount of familiarity with digital devices. It is in no way an attempt to explain how a computer works. The target audience is a card designer, who is perhaps familiar with another bus. Even with this assumption of a knowledgeable audience, some terms need to be defined. A word of warning to the reader: I am not a tech writer, and not even much of an author, I attempt to use these terms as defined, but occasionally I slip up. Please point these areas out for future correction.

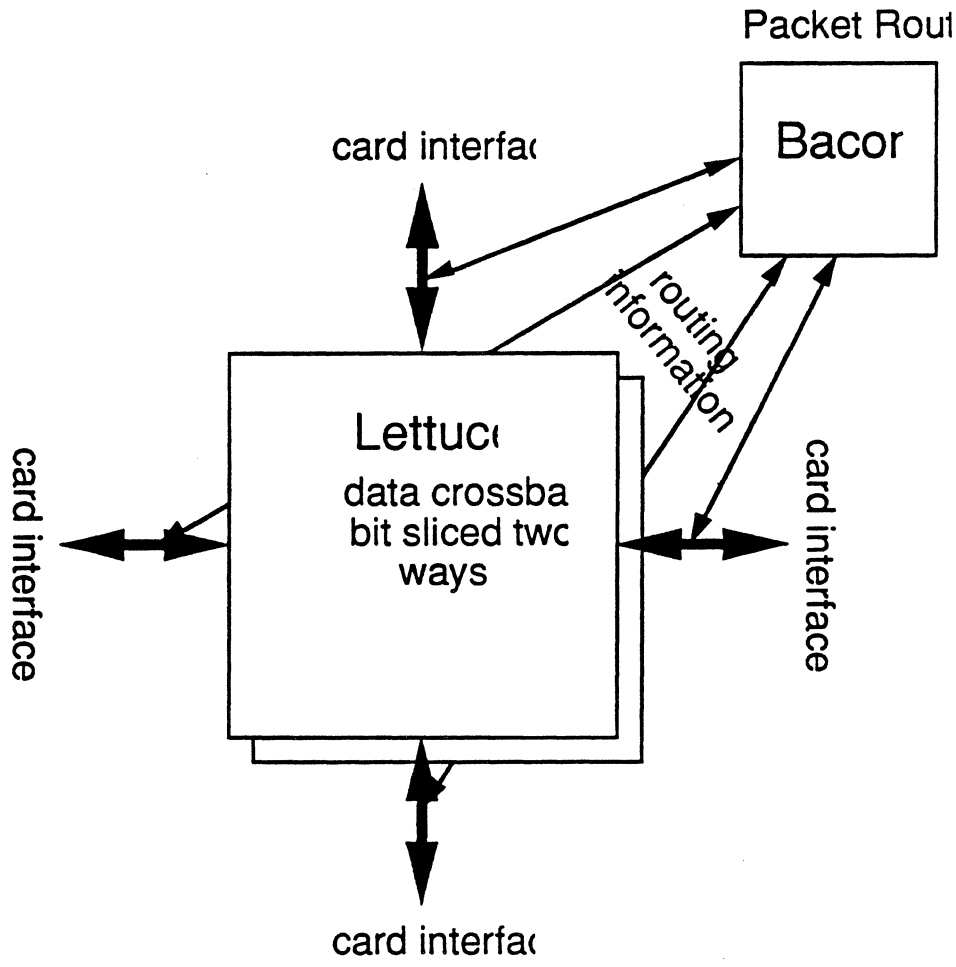
Table 1-2
Basic Definitions

Acknowledge	The act of asserting the acknowledge bit. This is the fundamental method of informing BLT that an operation has been accomplished
Card	A single board or device that is connected to BLT. Exactly what the name implies. Often used interchangeably with node.
Cycle	Describes the operation that must take place to handle a packet. Equivalent to a transfer for a write, one half of a read transfer.
Dump And Run Write	A descriptive name for a simple operation. An operation in which the write is completed into a local buffer before it is actually accomplished at its eventual destination. Heavily used by BLT.
Header	A set of twelve important data bits that describe a packet to BLT. The header is used by BLT to determine what to do with a packet and when.
Master	A device initiating a read of a write operation. A master is also termed the source of an operation.
Node	A specific port on BLT. Also used interchangeably with card.

Packet	The fundamental unit of data transfer on BLT. It consists of three types of information. The address which informs a card where the access is to, data which is the meat of the packet and a header. Packets contain from 0 to 64 bytes of data.
Slave	A card which responds to an access from another card. A card can be a slave only or a master and slave card. A master and slave card can both conduct transaction and respond to them.
Tick	The activity that take place during a single period of the clock for a node.
Transaction	The complete set of operations that are accomplished as a unit. In a write this is equivalent to creating a packet. For a read it is the entire operation from creating the read to getting a response back. In a locked operation, sometimes used to define everything done while the lock bit is asserted, many normal transactions.
Transfer	A single packet operation. Used to define complete act of loading, launching and receiving a packet. The operations required to send a packet through BLT.
Word	The unit of data transferred in a single clock period on the interface. This varies with the bus width, so in 16 bit mode a word is 16 bits, in 64 bit mode a word contains 64 bits

BLT Architecture

Figure 2-1
BLT architecture diagram



The most unique aspect of BLT is its basic architecture. Instead of connecting all the cards together with a bus, BLT uses an active backplane. All card communicate with a silicon switch that sends the bus transaction on to the appropriate destination. All the other unique features of BLT stem from this fundamental difference.

The basic model of a transfer is similar to any other bus. A card puts out address and then data to write information. A card receiving a write receives the address and data, acknowledges them and does the requested transfer. The basic facilities of a bus are still there, and in almost the exact same form as in conventional systems. The similarity ends there.

The best way to explain how the backplane works is to follow a typical example, a write of one word of data. The sending card first indicates that it is doing a send by loading a header into the crossbar. This header contains such vital information as priority, packet type (a read or a write etc.)

and destination (normally just upper bits of the address). Loading the header also has the important effect of configuring the backplane for the transaction. Next, the address and data are written into the crossbar. It is important to note that the this information is not actually sent anyplace at this point. It is stored in buffers internal to the backplane. This allows this data to be written at any rate, independent of the destination. When all the data is loaded (in this case after 1 word of data), the crossbar is informed that the packet is complete.

Now the crossbar routes the whole packet of data and address to the destination card. That card is then informed that a transfer has arrived for it. When it is ready, the card gets the address and then the data for the transfer. The card circuitry is responsible for doing what is appropriate with the data (storing it in RAM or whatever).

The exact mechanism for how a card interfaces with this is covered in great detail later. For now there are several important points. All writes are done as two independent transfers, one into BLT and one back out of it. The source (the card doing the write) puts the data for the entire transfer into the crossbar. In doing so it creates a "packet". This packet contains address and then a variable number of words of data. It has related to it routing information in a "header". This packet is the fundamental unit of transfer on BLT. After a writing card has created a packet the transaction is done as far as the the source is concerned. BLT takes care of getting the data to the correct destination and doing all of the handshakes with that card.

The other half of the transaction is the destinations removal of the packet of data from BLT. It does this almost exactly like dealing with a Nubus burst transfer. An address is presented for the packet and then the data for the entire packet. The card must simply store this as appropriate.

This "packetization" has a number of significant advantages. It means that the bus automatically does "dump and run" writes, i.e. the source of a write completes before the destination has dealt with the data. This is a significant performance improvement.

More importantly, packets separate the source from the destination. The entire transaction is loaded, and then it is written. This allows the two ends to run at completely different speeds, and even have different bus sizes. This makes the bus interface extremely powerful and yet very simple.

There is another performance advantage to packetization. In many transfers several sequential words will be transferred rather than just a single one. There is no reason to transfer the address independently for each word of data. On BLT packets are up to 64 bytes long. Cache line transfers and DMA operations benefit most from this performance enhancement. There is other overhead including initialization and routing that is reduced by lumping data together into packets.

BLT is designed and optimized to write data efficiently. The rest of the Jaguar is designed to do far more writes than reads. The Wilson DMA system is fundamentally write based. It is designed to allow cards to conduct almost nothing but writes. This system, discussed for the motherboard in its own section, is extendable and is intended as a system wide standard for data movement. The existence of the Wilson architecture is expected to bias the entire system toward burst write transfers. BLT is intended as the best possible transfer medium for such an environment.

The BLT architecture is efficient and powerful for writing data. Read operation are less favored by the architecture. There is a fundamental latency in sending data through an active backplane. This is hidden on writes by the built in buffering. There is no way to hide it for read operations. The address must travel through the backplane, the data must be read, and the data must

be returned through the backplane. The fastest this could be accomplished is about 200 ns which is exactly comparable to Nubus. In practice BLT will be faster because of wider data busses, a more efficient card interface, and higher clock speeds. The fundamental point remains that BLT is not great at read transactions.

There are other advantages to the crossbar architecture. Most significantly multiple cards can conduct simultaneous transactions. Internal to the backplane each card is directly connected to every other card. This means that if several transfers use different resources they can all be conducted simultaneously. For instance, card 1 could be sending data to the motherboard at the exact same time card 2 was sending data to card 3. In a normal bus these operations would be serialized by the bus. In BLT each card has a separate connection to every other device, so there is no contention for the backplane.

BLT supports eight levels of priority. The priority of every packet is set when it is created. This priority controls when the packet will be transferred relative to all other packets destined for the same card. If card A sends a packet at priority 2 to node C at the same time as card B sends a packet of priority 6 to node C, the high priority packet from card B will be handled first. This priority is absolute, so if while card C is dealing with the first packet card B sent another priority 6 packet it too would be received before the packet from node A. If node B continuously sent high priority packets to node C the packet from node A would never be received. Note that if card A were sending to any other node it would complete without regard to the transaction from node B since they would not collide.

There is a very interesting effect of priority. Consider the above node A with a low priority packet that is not being serviced. If that node were to send a priority 7 packet to node C an interesting event would occur. The priority 7 packet would be the next one serviced, before the "previous" priority 2 packet. This is called a loss of order. It is essentially required to support true priority, since a higher priority packet should be serviced first, regardless of the originating node. **BLT only defines order within a priority level for transactions from a single node to a single node.** If node A sent to B and C, there is no telling which would complete first. If node A sends a priority 7 packet and then one of priority 2 to node B, there is no order defined. However if node A sent two priority 4 packets to B, they are guaranteed to be conducted in the order sent. This order change is very different from the behavior of a conventional bus.

Card Interface

The BLT card interface has two very important goals. First it must be capable of handling data rates greater than 300 Megabytes per second. Second it must make a simple card cheap and trivial to implement. These two goals are almost mutually exclusive. The first demands wide busses, fast clock rates, and probably would not balk at requiring a gate array for implementation. The second demands small busses, slow clock, and if it can't be done in a few PALs it will never exist. To address these two extremes an entirely new bus interface was created for BLT. Instead of trying to create one interface that met the diverse needs of card designers, BLT adopts a polymorphic interface that changes shape to fit the specific demands of a card.

There are 5 fundamentally different bus interfaces for BLT. Two are aimed at gate array implementations. These feature multiplexed address and data for pin efficiency. The first is a 64 bit

interface. It is intended for high performance devices such as frame buffers and high speed processors. The other multiplexed mode is 32 bits wide. It is roughly equivalent to Nubus. It is mainly intended for gate array implementations of medium level performance. Good examples of this sort would be 68020 or higher cards, or a video input card.

The other three card interface modes are non-multiplexed. 8, 16 and 32 bit busses are supported. All three have a separate 32 bit address bus. The address is automatically incremented to keep it consistent with the current data. These interfaces will be used by the bulk of our third party developers. They range from medium to low performance (200 MB/sec and below... medium performance). The control signals are simple enough that a 2 or 3 PAL interface is possible.

Bus width is only half the battle in building a simple card interface. The other problem is clock speed. It is rather silly to have to build a GPIB card with a 50 Mhz bus interface. If a fixed frequency is used by the interconnect it virtually guarantees that synchronization will have to occur to get to the frequency the card actually uses. This induces a significant delay in many systems. To avoid both these problems the BLT interface is run synchronous to a clock fed to it by the card. This allows the GPIB card to run at 2 Mhz, the frame buffer to run at 40 Mhz, and the processor to run at 28.3 Mhz. To avoid burdening all cards with a crystal a system standard clock is available to the card for optional use.

All of these different interfaces can freely and invisibly cross communicate. A 64 bit word sent from the motherboard becomes 8 8 bit transfers to the serial interface card. 8 8 bit transfers from the serial controller become one 64 bit transfer to the motherboard. This capability places one demand on cards: support for burst transfers. This is much less difficult then it sounds. In the separate modes the address is always kept consistent with the data. The bus itself indicates when the last "word" is about to be finished and when the transfer is done.

The bus is fundamentally a byte stream device. All addresses are for byte alignments. A transfer can be an arbitrary number of bytes long, up to 64. In fact, there is no such thing as a non-burst transfer. All transfers are treated exactly the same independent of length. A receiving card does not know how long a transaction is until it has reached the end.

In further support of byte streams there are no byte lane constraints. Built into BLT there is a byte twister. This allows any byte to be brought out (or sent in) on any byte lane. This is primarily targeted at DMA devices. It allows byte justified addresses independent of the bus width. An example of this is a 32 bit device that is told to do a burst transfer that starts at address 3. In Nubus this would have to be done as several transfers (including one byte twist). In BLT each byte can be loaded on its "natural" byte lane. The backplane will handle twisting it around until it is appropriate for the destination (which may want it twisted to someplace else).

In addition to these new features BLT features several conventional niceties. The interface is fully synchronous to the card clock. Unlike Nubus, both source and destination can stall a transfer at any time.

The BLT interface is as generic as possible. It cleanly supports cards whose performance is slow for an Apple II as well as those whose bandwidth that would swamp the Cray. It demands as little as possible from the card. It is simpler, more powerful, and cheaper than interfacing to Nubus.

Streams

So far the only operations that have been discussed are burst reads and writes. Both are available in a conventional interconnect although harder to use. The stream is a data transaction that is new to backplanes. The fundamental building block is quite simple, a flow controlled write.

The entire Wilson system is built around the concept of a stream. It is simply an address-free stream of bytes. It is used to transfer data between two data processing elements. It is a concept that is fairly old to software folks, its most common embodiment being unix pipes. It is a context free method of routing data. The only thing that is tricky about it is that it must have flow control.

Consider two independent sources sending data to a single destination. The destination does the simple act of taking one byte from each and summing them, and then send the result on. This is an operation that is almost impossible on a normal bus. What happens when source 1 is running a little bit faster than source 2? At some point the destination is full of source 1 data and is about to receive another byte from the same source. It cannot get rid of any of the data because it needs data from source 2 to do that. But source 1 owns the bus, trying to send another byte of data. Source 2 cannot send the data needed to consume data until some data is consumed, a deadlock. This application may seem a little esoteric, but it is the exact operation required to fade between two video sources using Wilson.

To avoid this a method of blocking must be implemented. The destination must be able to refuse data from source 1 until it has its data from source 2. This is the fundamental operation necessary to support a stream of data. BLT supports it cleanly and efficiently.

Internally BLT buffers all information. The packet is loaded first, then it is routed, then it is fed to the destination. Normally the card has no choice about which packet it will receive next; the decision is made internally based on priority. If a card supports streams this is changed. The card can look at the header of the packet and then refuse it. This implements the destination flow control of writes.

Stream support also impacts the source half of a transfer. BLT normally supports a limited sort of flow control. If all the internal buffers are full a card cannot send more data. In stream mode this is extended. A card can determine if a specific packet has been transfered. This allows a sending card to support multiple independent streams. For instance it could be sending one stream to the motherboard and one to the frame buffer. Normally if the motherboard blocks the stream all the buffers would fill and the other stream would also block. This must be avoided. Instead of blindly loading more and more packets to be sent to the motherboard, the card checks to be sure the previous one has been sent. If not it does not load a new one. This prevents a single stream from stopping the entire card.

Section 3

Card Interface

This section discusses the signals and timings involved in the card interface. The goal of the card interface is inexpensive implementation, which means as few components as possible. The interface provided here is a second cut, but still needs significant refining. Whatever is eventually specified will be a piece of the architecture that we must live with for the life of the product line.

Basics

The card interface is as simple as possible. Since the basic function is the same as for a conventional bus, the basic structure of a transfer is the same. However, the mechanism differs extensively in the details. The purpose of this section is to give a quick overview of the basic methodology of a transfer.

There are two important ideas that BLT is based upon, packets and headers. A packet is the fundamental unit of data transfer. It always contains an address. It can contain from 0 to 64 bytes of data. Internally, BLT only knows about packets. They are stored as a block of data with an address at the beginning. The card interface portion of BLT is responsible for converting to and from this packet format. The internal guts move packets from node to node.

This packet model has some important impact on how BLT functions. Since all data transfers are via packets, it does not matter how much data is in a packet. The model for moving one byte is the exact same as the model for moving 27 or 64. This has two important effects. First, all devices must be able to deal with any size transfers. Secondly, BLT is independent of how large the card interface bus is. A packet containing 12 bytes of data implies twelve data transaction on a 8 bit interface and 2 (with the second a partial word) on a 64 bit interface. Both are capable of dealing with any size packet. This is why BLT can implement multiple interface sizes.

Loading a packet is fairly straight forward. A card first loads in the address for the packet. This address is always 32 bits wide, and is always loaded as the first tick of the transfer. The card then loads the data for the packet. In the separate modes, the first word of data is loaded with the address. When it has loaded all that in wants to (less then 64 bytes) it asserts the done signal. This indicates to BLT that the packet is complete.

Note that there is no restriction about how a card loads data. It could build the packet up one byte at a time. It could also send in 8 bytes per tick. A really bizarre card could send in a different number of bytes each time. The BLT card interface happily converts it all into a tightly packed packet. When the done signal is finally asserted, the packet is marked complete and is ready to be sent.

There is a critical piece of information missing at this point. BLT has no idea where to send the packet, what is in the packet, or how important it is. All of this information is contained in the header. The header is side information that tell BLT what to do with a packet. BLT does not interpret any information within a packet. It is entirely controlled via the header.

There are three important pieces of information in the header: node ID, priority, and packet type. The node ID indicates where the packet is to be sent. In Jaguar this is defined by the upper four bits of the address. The priority field indicates to BLT how important the packet is. This controls when the packet will be transferred. One of the unique aspects of a crossbar is that many transfers can occur at the same time. Priority allows the external system to have some control over which ones should go first during collisions. The final element of the header is the packet type. This identifies whether the packet is a read, a write, or one of various special packet types. In the event the packet is a read, the size of the block being read is encoded into this field as well.

The entire purpose of the card interface is to get packets and headers into and out of BLT. There are dedicated lines for loading the header. These lines serve two purposes. When a device is creating a packet these lines define the header for that packet. When a device is a slave and receiving a packet, these lines output the packet information, telling the card what to do with the packet.

To create a new packet, a card first passes the header to BLT. This entails asserting the header values on the header lines and then asserting the header valid signal. The new packet can now be loaded as described above. When the packet has been loaded and the done signal asserted, the card has completed the transfer. BLT takes care of getting the packet to the destination and feeding it out to that card.

Reads are conducted in a similar fashion. A card creates a new read packet with the packet type bits set as appropriate. It then loads the address to be read and asserts done. Now, just as for a write, BLT takes over and routes the packet to the destination. The header is presented to the slave, which tells the slave that the transaction is a read. The address is also presented. The card responds by loading the requested data into what is know as a read response packet.

A read response packet is almost identical to a normal write packet with one important difference; because BLT knows that the data is in response to a specific read, no header needs to be asserted. BLT automatically derives the appropriate header from the read packet. Because of the protocol, an address is automatically loaded, but it is meaningless for a read response.

Eventually the read response packet is loaded and the done signal asserted. BLT routes this packet back to the originating node. Here it is presented as any other packet. The header is output to the card, but the packet type is read response. This informs the card that the packet is the response to the read.

There are many additional complexities that can be added to a transaction. However, this basic model is maintained for all of them.

Signals

The card interface consists of 103 signal lines. These are broken into 4 logical groups. The data control line are the basic lines used in moving data. The header control lines are used to supply

or receive information about a packet. May of these lines are only used by cards that can perform master operations. The packet status lines contain special information about the status of packets. The final group of control signals are the system interface. These provide basic system functions such as clock and reset. Figure 3.1 lists all of the signals. The rest of this sub-section is dedicated to an explanation of these signals.

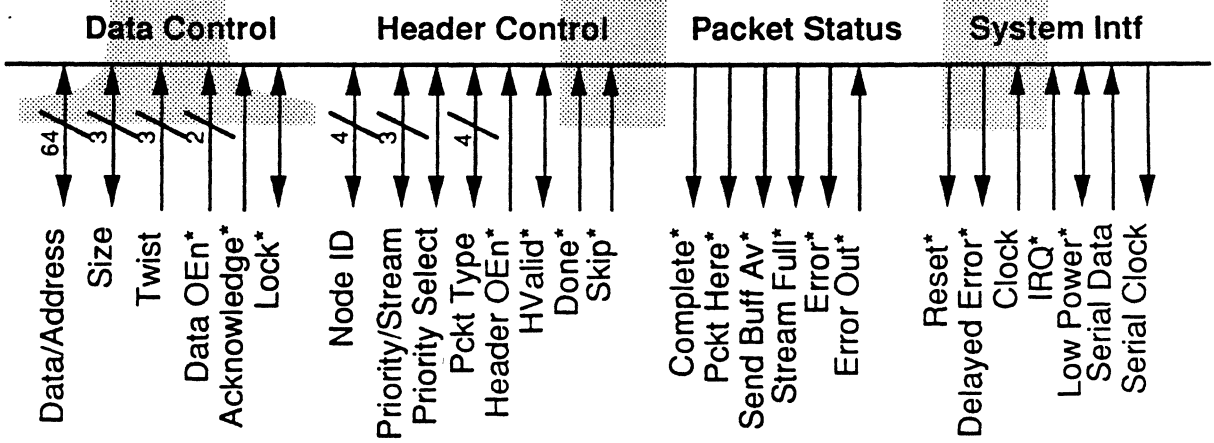
Almost all signals in BLT are synchronous. The only exceptions are the twist lines and the signals they affect: data, size, and complete. These later signals are changed synchronously by BLT whenever BLT induces that change. They are combinatorial only to changes in twist by the card. The interrupt and reset lines are also asynchronous.

All synchronous signals are sampled on the rising edge of the card clock. They must meet a defined setup time to that edge. All synchronous signals are driven off the rising edge.

An integral part of BLT is a serial bus. This bus is used to accomplish initialization and error recovery. It is not intended as a card to card communication medium, although it can be used as such. It provides a mechanism for accessing internal state of BLT. This is required to make recovery from write errors possible.

In the description that follows, two conventions are used. Active low signals are specified by a trailing asterisk. All signals are labeled as being input, output, or bidirectional. The definition of input and output is from the card point of view. All output signals are driven by the card, all input signals are driven by BLT.

Figure 3-1
BLT Card Interface Signal List



Data Control

The data control signals in some sense perform all the work of BLT. All data and address move over these pins. There are two independent output enables, one for each 32 bits of the bus. In the modes that have separate address and data busses, the output enables allow either to be controlled independently.

The twist and size lines are something not seen in any other bus. These lines control the internal barrel shifter in BLT. This allows transfers to be aligned to any byte boundary. The twist lines indicate how the data will be presented, the size lines indicate how much is valid. These lines make many common interface problems much simpler.

The acknowledge signal is the most important in all of BLT. It is the signal that indicates when any data is being transferred. To load data, assert it on the data lines and assert the acknowledge line. To receive data, assert the acknowledge pin. The acknowledge pin allows a card to control when BLT transfers data. If acknowledge is held asserted, data is transferred every tick, if it is never asserted, nothing happens.

Data<31:0> (bidir)

The lower half of the data bus is always used for data transfers. The least significant bits possible are used for any given mode. In 16 bit mode bit <15:0> are used. In 8 bit mode bit <7:0> are active. In 32 bit multiplexed mode these pins are not used. In 64 bit multiplexed mode these pins contain the least significant 32 bits of data (byte lanes 3-0). In the first tick of a 64 bit multiplexed cycle, when the upper bits contain address, these pins are ignored as outputs and zeroed as inputs.

Data<63:32>/Address<31:0> (bidir)

The upper half of the data bus is used for two different purposes. In all modes it serves as the address bus. In the multiplexed modes it also is the data bus. In 64 bit multiplexed mode it contains the upper 32 bits, (byte lanes 7-4). In 32 bit multiplexed mode it contains both address and data in turn.

One note of importance is the address bus lines up differently from the data bus to the pins. The connections are as follows: Data32-Address0, Data33-Address16, Data34-Address1, Data35-Address17... Data62-Address15, Data63-Address31. This twisting is necessary to allow the appropriate internal chip partitions for address incrementing.

Twist<2:0> (outputs)

These 3 bits specify the starting byte lane for a specific transfer. They specify the address the least significant byte of the next word is on. Note that since BLT is big endian the address of a byte lane is the inverse of the byte lane number. In a 64 bit interface, byte lane seven is address 0. They are active both for output to BLT and input to BLT. They allow arbitrary byte alignment.

Most commonly the twist lines are connected to the low order address bits. This allows efficient byte aligned burst transfers to be easily supported. Consider a 64 bit transfer to a 32 bit card with a starting address of 2. This means that the first byte of the transfer should be placed at bits <15:8> (big endian) on the card. By connecting the address line to the twist bits the appropriate rotation will be accomplished. In all cases of twisting on BLT input, the size bits will be adjusted to write as many bytes as possible without wrapping around. Thus in this case BLT will output a size of 2 for the first tick. This will have the affect of zeroing the address bus for the next and all successive ticks.

The twist bits are particularly useful for DMA transfers. In the specific case of Wilson window transfers the twist bits are used to twist incoming pixels into the appropriate byte lane to align to a window.

The data bus, size bits, and complete signal are all affected by changes to the twist lines. These changes are purely combinatorial. Care must be taken to allow time for these transitions to propagate through. Specifications will be provided.

Any twist bits that are rendered meaningless by bus width (bit 2 in 32 bit mode, bits <2:1> in 16 bit mode, all in 8 bit mode) are ignored by BLT on card output.

The address is never effected by the twist bits. It always appears and is provided on the upper half of the data bus.

Size<2:0> (bidir)

The size bits indicate how many bytes are being moved on the current tick. The size is always based upon where the twist lines define the starting byte lane to be. Note that in some cases the size bits can indicate a wrap around the end of the bus. For example in 64 bit mode, size equals 8, twist equals 1, puts the eighth byte back in byte lane seven, a bus wrap. This is permitted on output to BLT but will never be generated by BLT on input. Care should be taken that wrap is really what a card designer intends.

On input to a card BLT will always try to transfer as many bytes as possible without causing bus wrap. Note that since the length of a packet is arbitrary no assumptions about the size for any tick can be made.

The size lines may be set to anything during output from card. BLT will convert multiple small loads into larger words before transfer.

The size bits indicate the number of bytes being acknowledged in a given tick. The address in a separate mode is incremented by the number of bytes indicated by size after an acknowledge.

Any size bits that are rendered meaningless by bus width (bit 2 in 32 bit mode, bits <2:1> in 16 bit mode, all in 8 bit mode) are ignored by BLT on card output. They are set by BLT when it is inputting to a card. This prevents "overlap" on bus wrapping.

The direction of the size bits is defined by data output enable 0, so it is consistent with the data bus in the separate modes. In the the 32 bit multiplexed mode it is possible to screw with this relationship since the upper half of the data bus is used. It is strongly suggested that for this mode the two output enables be tied together.

Figure 3-2
Byte Lane Explanation

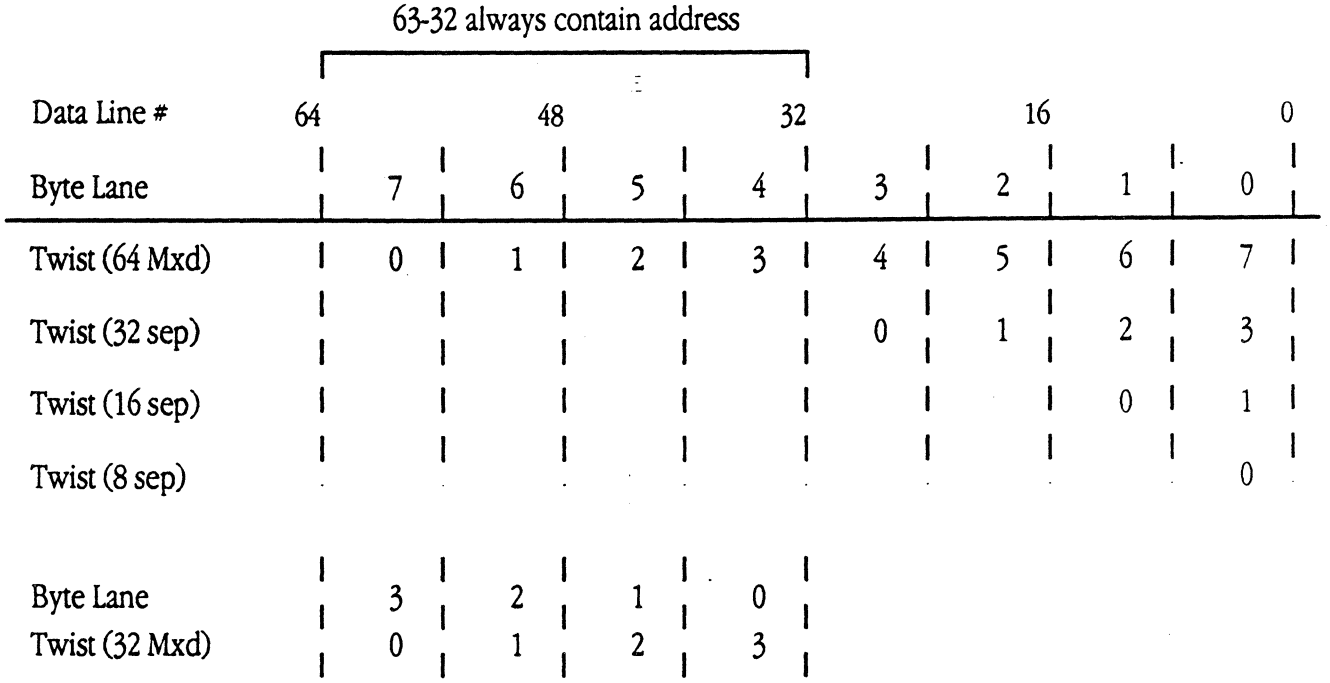


Figure 3-2 shows that byte lanes for the various interface mode. For all modes except 32 bit multiplexed the byte lanes are consistent with the data line signal numbering. An 8 bit separate interface uses pins data<7:0> for its data bus. A 32 bit separate interface uses data<31:0>. The twist lines specify the address, not the byte lane. For an aligned transaction the twist lines are always zero. For these modes, the same effect as the differing modes could be achieved by asserting all unused twist lines high. This is what BLT does internally.

In 32 bit multiplexed mode, the data lines are moved to the high order 32 bits. This allows that address lines to always remain on the same pins. The byte lane definitions also change for this mode.

Lock* (bidir)

The lock bit is a rather bizarre signal. When it is driven by BLT it indicates that another card is conducting locked transactions on the local card. It is driven to BLT along with the address to lock a remote node.

The lock bit is asserted to a slave card upon the reception (acknowledge of the address or first data word) of a locked packet. It indicates that the card should take whatever actions are necessary to lock the local resources. At a slave the lock bit is deasserted with no association to a transfer. That is the lock bit just goes away at some point. This indicates that the locking card is done with its transaction.

It is possible for slave to send a new packet while it is locked. In order to accomplish this it must turn the data and address bus around. This makes the lock bit an input. This does not imply that the slave is no longer locked. The slave must maintain the current lock state while it is sending to BLT. The lock of this card can only be released when the lock bit is driven to one by BLT.

On the master side locked transactions are fairly easy to accomplish. The first access is simply conducted with the locked bit set. The lock bit must remain asserted for as long as the lock is desired on the remote card. As soon as the lock bit is deasserted when not actively in a transfer, the remote resource will be freed. If the lock bit is deasserted in the midst of a transaction (loading a packet) the entire packet will be transferred as locked and then the remote card will be unlocked.

The lock bit is a bidirectional signal. If during a remote lock the Data Oen1 is asserted the current state of the lock bit is sampled. The external resource will remain locked until the address bus is asserted back to output with the lock bit low. This means that the lock can be maintained by simply having the lock bit asserted when Data OEn1 is deasserted (watch for contention).

BLT internally flushes all buffers before conducting the locked transfer to the remote card. A locked packet is therefor conducted in order to all previous transfers. Within the lock itself no such rules apply, so the entire lock should be conducted at a single priority level to guarantee the expected results.

On the slave side BLT will guarantee that no other accesses will be presented from any other node until the lock is released.

BLT does not interpret the lock bit on the master side. Transactions will still be presented to the card for completion and must be dealt with. See the abort section for more details on these operations.

Data OEn<1:0>* (outputs)

The two data output enables control the direction of the data and address busses. Data OEn1 controls the upper 32 bits/address bus and the lock bit. Data OEn0 controls the low order bits of the data bus (the entire data bus in separate modes) and he direction of the size lines.

These signals tri-state the outputs immediately (combinatorial), but are used synchronously to change the direction of the bus. This provides a dead tick on all bus direction changes to avoid bus contention.

Acknowledge* (output)

The acknowledge signal is used to inform BLT that a specific word transfer is complete. In general it indicates that size bytes of data have been transfered. The only exception is on multiplexed transactions, in which the first acknowledge transfers address, while the second and subsequent "acknowledges" transfer data on writes. In separate modes the first acknowledge transfers address and data, while subsequent ones transfer only data.

BLT is capable of single tick transfers. Holding acknowledge asserted has the effect of transferring one word every clock period. There is no rate limit to how fast acknowledges have to be asserted, except that timeout constraints, as specified later, must be met.

Acknowledge is a purely synchronous signal. It is sampled on the rising edge of each clock period and acted on accordingly.

Header Control

The header control section is used to define and interact with information related to a packet which does not directly impact the data transfer, the header. The header control section has three sets of bidirectional pins: node ID, Priority/Stream, and Pckt Type. These pins are used by a master to define important parameters for a packet. They are driven to a slave device to pass the same information on. They are independently loaded and controlled from the data interface.

When starting a transaction a master device may load header, address, and data all at the same time or independently. Internal to BLT there is one pipeline register that is loaded with address

or data by the acknowledge. This register is only unloaded once a valid header has been loaded. Thus the header must be valid before the second acknowledge is asserted for a master write. Note that a master read can be launched in one tick by asserting header, address, header valid, and done all simultaneously. A small write can also be launched in a single tick.

The header control bits are also used to carry card configuration information during system initialization. These signals should be tied with pull up/down resistors to the value appropriate for the card. These values are sensed during reset. This information is then used to configure the interface as requested. Care must be taken to insure that these signals will be at their appropriate values during this time. This usually entails ensuring that all driving devices are tri-stated during reset. BLT guarantees this for all its signals.

Node ID<3:0> (bidir)

The node ID is used to identify which node a transfer is to or from. On a master operation these bits must be driven to the number of the node a transaction is destined for. For a slave operation these bits are driven to the node a transaction is from.

In Jaguar the destination is defined by the upper four bits of address. BLT handles the transformation from the 16 possible combinations to the address map appropriate to the machine. Thus it is presumed that the node ID will be driven to these bits when a header is initialized.

In a simple master card it is likely that the upper bits of address will be connected directly to these pins. Care must be taken for slave transaction if this is done, since the value asserted on the node ID lines is the originating node, which has no relation to the destination address (except being guaranteed to be different). The header outputs must be disabled before the address is enabled.

Priority Select (bidir)

This bit determines whether the packet is a member of a stream transaction or a conventional transfer with priority. If it is high then the transaction is conventional.

Priority and streams are in some sense mutually exclusive concepts. The concept of priority allows important transfers, as defined by the source to bypass less important transfers. This is implemented by controlling which packet is presented to the destination. Streams on the other hand are sorted by the destination. It looks at all the streams and decides for itself which one is "important" at this point. By their very nature streams fly in the face of priority, because higher number streams are exactly equal in "priority" to lower number streams.

In implementation there are two fundamental differences between streams and priority. The first is that as mentioned above streams are not sorted by number. The second is that in order to support priority promotion is necessary.

Promotion handles the case of a node entirely full of low priority packets wanting to send a high priority packet. Since the high priority packet cannot be put into BLT it can never arbitrate. Therefore it is effectively arbitrating at the low priority level of the packets ahead of it. To fix this, when this situation occurs one of the low priority packets is "promoted". It is raised in priority to the priority of the new packet, allowing the high priority packet the attention it deserves.

Earlier it was mentioned that order was not maintained between priority levels. In the special case of a packet being promoted it still maintains order with other packets of its old priority level. This is accomplished by selecting the proper packet for promotion.

Streams are identified by their stream number. Obviously if a stream packet were promoted there would be a problem since it would suddenly become a member of a different stream. This is why streams and priority must be separated.

In arbitration, streams are considered the lowest priority. If there are any priority transfers of any level pending for this node, a stream transfer will not be presented. See the Skip signal for how this can be gotten around.

This bit is often lumped into a term called priority/stream. In most instances this term is a reference to four bits, the priority select bit and the priority/stream bits.

Priority/Stream<2:0> (bidir)

The priority/stream bits serve several purposes. If the priority select bit is a one then they specify the priority of the packet. If the select bit is low then they specify the stream number.

Much like node ID, these bits are set by a master and received by a slave. The only difference is that when priority they can change once they have been presented to a card. This will occur if a packet is promoted while it is the current one about to be output.

Most slave cards can ignore priority. The only time this will not occur is if further arbitration on card is necessary. Master cards will set priority as determined by software.

Streams are handled almost identically to conventional packets. These bits simply define which stream the current transfer is for. Streams never promote, as there is no priority defined.

An additional use of these bits is for querying about the status of a packet. This is discussed with the Stream Full bit.

The two low order bits are used to configure the abort response. Bit 1 defines the lock abort mode and bit 0 defines the read abort mode. The exact interpretation of these signals is discussed later in this section.

Pckt Type<3:0> (bidir)

The packet type bits determine what the current transaction is. There are three basic packet types: read, write, and read response.

Table 3-1
Packet Type Encodings

0000	**Read Response Retry:** this is used to indicate that the read must retried after the local bus is given up for arbitration.
0001	**Read Response Address Error:** the read that this response is related to is to an invalid device. This should be dealt with as a bus error for most processors.
0010	**Read Response Card Error:** the card this read is from has indicated an error in responding to the read. The card firmware should deal with cleaning this up.
0011	**Read Response Card Timeout:** The card this read is from is not responding. The card firmware should deal with this problem.
0100	**Read Response:** used to specify a normal reply to a read operation. Usually only seen by a master for the response to a read. Driven to BLT during a slave split response transaction.
0111	**Write:** A standard write packet.

1nnn	**Read:** A read of a specified size block of data. nnn specifies the size as a power of two in bytes. 1000 specifies a 1 byte read. 1110 specifies a read of 64 bytes.
1111	**Not Used:** This coding is not used and is reserved for future use.

A simple slave card does not need to decode any but the uppermost bit. On read operations the complete line indicates when the proper amount of data has been transfered. This allows a card to simply continue feeding data back until the complete signal is asserted.

The three low order bits are used during reset for configuration. These determine the card interface type and size: multiplexed or non-multiplexed, and the size within that. The exact definition of these bits discussed later in the initialization section.

HValid* (bidir)

The header valid signal serves two functions. When driven to BLT it initializes a packet. When driven by BLT is defines when the packet control bits are valid.

When creating a new packet, header valid is asserted when the header data is valid. If send buffer available is asserted, then a new packet is initialized. If there is valid data in the pipeline register it is loaded as the first word of the packet.

When BLT drives HValid it signals that a valid header is presented on the lines. This does not imply that the transfer associated with that packet is ready. It is simply an early signal indicating valid header data, which is very handy for rapidly skipping through stream headers.

Header OEn* (output)

This line controls the direction of the header control signals: node ID, priority select, priority/stream, packet type, and header valid. If it is asserted the header is driven by BLT. If it is deasserted BLT is receiving from the card. This bit has no effect on any current transactions. Its direction can be changed at any time, including while a transaction is going on. This is intended to allow a card to do some pipelining, for example selecting the next input stream while launching a packet.

Exactly like the Data OEn signals these tri-state the outputs immediately (combinatorial), but are used synchronously to change the direction of the bus. This provides a dead tick on all bus direction changes to avoid bus contention.

Done* (output)

The Done signal indicates that a packet has been completed. It is asserted by the card during the last acknowledge. This indicates to BLT that the packet should be routed to the destination. For slave accesses, this signal need not be driven.

The done signal can be used by a slave device to perform true split response reads. When a read packet is presented to a node it can be dealt with in two ways. Either the data is presented immediately or a split read is initiated. In a split read operation the read packet is artificially disposed of before the data has been fed back. At some later time the node is responsible for driving the appropriate data.

A split read is initiated by responding to the read packet with a done signal before any data has been returned. It is presumed that the card has latched whatever data it needs to respond at a later time. When the card is ready to respond to a read, it generates a read response exactly as it would generate a write packet. It must load a header with packet type read response, node ID equal to the read originator, and priority equal to the read packet.

The split read operation is fully discussed later in this paper. It has some dangerous aspects that a card designer must be aware of.

Skip* (output)

This is the fundamental mechanism for destination control of streams. It indicates to BLT that another packet should be presented for the card to receive.

BLT will round-robin through all streams that currently have packets waiting to be received. It will not present any other packets from the same stream. With sufficient skips, the original packet will be presented again. Note that order within streams will be guaranteed since skip will always present the same packet from any given stream.

For purposes of the skip pin, all priority transfers are considered part of a single stream. That is if a priority packet is skipped no other priority packet will be presented again until all streams have been presented. Note if a priority packet is skipped, and then the entire set of stream packets is skipped, the original priority packet may not be presented again. If a higher priority packet is received in the meantime it will be presented instead.

Packet Status

The packet status bits are used to transfer information about the interconnect status. The header and data control signals are exclusively generated by a card. They are routed and manipulated by BLT, but it does not create them. The packet status bits are exclusively created by BLT, and exist only for interfacing with BLT.

Complete* (input)

The complete bit indicates to the card that the current word is the last word in a transfer. It is active for all slave transactions and for read responses. It is asserted as soon as BLT determines that an acknowledge will complete the packet. If an acknowledge is given when complete is asserted the packet is done. The complete bit is combinatorial with respect to the twist and size lines. If these are changed the complete signal may change as well. Care must be taken to allow sufficient time for complete to stabilize before asserting acknowledge since it is also used internally by BLT.

Pckt Here* (input)

The packet here signal indicates that a packet is ready to be delivered to a node. If the data output enables are asserted it also indicates that the first word of the packet is valid (address and data in separate modes, address in multiplexed modes).

The packet here signal will remain asserted until an acknowledge is driven to BLT when the complete or done bit is asserted. If there are no other pending transaction, it will be deasserted by the next rising edge. If there are pending transaction it can remain asserted indicating the start of another transaction.

Send Buff Av* (input)

This bit indicates when a buffer is available for a master device to originate a transfer. If this bit is asserted, a device can create a new packet by asserting the header valid signal.

If this bit is not asserted it indicates that all the buffers in BLT are full. If a card still asserts the header valid signal, a promotion occurs if necessary. On the first tick that send buffer available is asserted along with header valid a new packet is initialized.

One exception to the above rules is read response packets. In order to avoid a deadlock, all read responses use an independent buffer. The send buffer available bit does not reflect the status of that buffer. If a read is presented to a card, the read response buffer is empty so a read response can be generated.

Stream Full* (input)

The fundamental capability of streams is that they can be blocked by the destination. In order to avoid this totally filling the sending cards buffers it is necessary to pass this information out to the card. The stream full bit allows that functionality.

The stream full bit combinatorially indicates if there is a packet waiting to be transfered at the currently asserted priority/stream number. If there is a packet this bit is asserted. If there is not a packet this bit is deasserted.

This allows a sending card to query BLT on the status of a stream. If there is a packet currently waiting, the card need not send another because the stream is blocked. If there is no packet in BLT the stream has been serviced so another packet can be loaded.

Stream full is not a control signal, it simply carries information that allows a stream protocol to be implemented. The restrictions of send buffer available must be obeyed.

Error* (input)

The error signal indicates that the node addressed by the current write packet does not exist. This is almost always due to an address error in software.

On a write operation the error signal is asserted the tick after header valid if a problem exists. It remains asserted until either the packet has been completed by a done signal or a valid header is loaded. If the packet is already completed the error signal remains asserted for one tick. In any case the entire packet is tossed. Note that everything works if a new packet is started without completing the errored one.

A read error is handled a little differently. It seems easier for a card if a read always generates a read response packet. For this reason a read will generate a header with an address error packet type if a bad read is launched.

Error Out* (output)

The error out signal is used by a slave device to indicate that a packet cannot be completed due to an error. It can be asserted at any time before the packet is complete, either on a write or a read. The response to the packet should not be completed. The packet is immediately terminated by removal of packet here and header valid.

System Interface

These signals provide the basic elements that a card needs. Most of these signals have very obvious functions, and exist in any bus. Some, such as delayed error and low power are a little more esoteric.

Clock (output)

The clock signal defines the clock that the board interface runs synchronous to. All transitions are based upon this signal

The clock will have a minimum operating frequency of 100 KHz and a maximum frequency of 60 MHz. These number will be adjusted based on reality. One possibility is to allow the clock to be totally static during power down mode as a method of reducing power. This will work, but a constant clock is necessary during operation.

This signal is presumed to be asynchronous to both the system clock and all other board clocks. All signals are fully synchronized before they are passed to any other node. There is no performance advantage to using any specific frequency.

IRQ* (output)

An interrupt line connected directly to the Mazda I/O processor. In provides a method of simple interrupts. The exact interrupt architecture is defined in the I/O section of the ERS. The interrupt is defined to be edge sensitive. The falling edge of the signal will generate an interrupt. The exact effect of this signal is discussed later in this paper.

This signal has no necessary relationship to any clock. It will have to meet minimum high and low times.

Reset* (input)

A card reset line. This signal will be independent for each card to allow total board reset independent of the rest of the system. If a card is not sensed present this signal will be tri-stated to coincide with the live insertion/removal model covered in the physical section.

Delayed Error* (input)

This signal indicates that a packet that was written by this card has encountered an error before completion. This will only occur because of a card asserting an error on the packet or because the packet timed out, indicating total board dysfunction.

Recovering from these errors will be particularly difficult, because the transaction was completed sometime earlier by the writing device. The serial bus is used to access the corrupted packet, as well as to clear the buffer when recovery has been accomplished.

This signal will usually be used to generate an interrupt to whatever device is responsible for error handling. For simple cards this will be the motherboard. Complex cards may wish to do their own error handling.

Low Power* (wire or)

The low power pin indicates that the machine has gone into low power mode. In this mode all non-essential processing should be halted and the power utilization of the board should minimized. The low power signal will go low when all cards and the motherboard agree that it is appropriate. In addition to this line there will be a firmware call in the card ROM definition to allow arbitrary software actions upon entering this mode. Note that the firmware is called before the low power signal is asserted.

The intent of low power mode is to reduce the power consumption of the system sufficiently to allow any fan to be shutdown. The hard disks will also be spun down in low power mode. This will allow Jaguars to remain on all the time. Appropriate power saving actions are halting processors, stopping RAM access, turning off video, etc. The machine should appear off in this mode, with the exception of phone, sensors, or other wake type actions. The only time a Jaguar should be turned off is if somebody is stealing it (or moving, but that is boring).

Any card can prevent or recover from low power mode by pulling this pin high. This will restart the entire system, spin up the disks and make all system resource available. There will be some time lag between pulling this pin high and full system operation. There is a firmware vector defined for informing a card that the system is fully alive.

In general, a card should not prevent the system from going into low power mode. The motherboard should be allowed to make this decision. Of course, the card can prevent it if necessary, for instance actively transferring information. The best approach when possible is to let the system power down whenever it wants, but bring it back up when necessary. A Jaguar should spend a large chunk of its life in low power mode.

Serial Data (bidir)

This pin gives access to the serial bus. The protocol is not yet determined. The serial bus is intended for system initialization and error recovery. Most devices will not connect to it. Jaguar provides an address space that is decoded into serial bus transactions, so access can be gained that

way if desired for master only transfers. It is not required that any device connect to these pins, although if a device wishes to do its own error handling it probably should.

The principle goal of the serial bus is to solve a system problem of configuration and error recovery. It is made available to cards for two reasons. First, some card may need direct access to the error recovery mechanism. Second, it can also serve as a simple card to card communication mechanism. The value of this last is small, as BLT is quite simple to use. The bottom line is this access is cost free, so make the system as flexible as possible.

Serial Clk (input)

This signal serves two purposes. First, it provides a guaranteed 10 Mhz clock. This will alleviate the need of an expensive oscillator on many cards. Second, it is the data strobe for the serial bus. The protocol of the serial bus has yet to be determined, but most involve a constant clock and a related data line.

Initialization

There are five pins that are used for system initialization. These are piggy-backed on top of some of the header control signals. These signals are sampled during reset before the reset line goes high. The board must guarantee that these signals are at the appropriate levels during this time. The sample time is such that the power supply has been stable for n milliseconds. All the signals used are bidirectional so the additional overhead of providing this configuration information is minimal.

One way of achieving the proper configuration values is by use of 1k pull up and pull down resistors. The only additional requirement is that all drivers are disabled during reset. BLT will guarantee that its drivers are off.

The two low order bits of priority/stream are used to configure the retry response. The zeroth bit is known as the read retry signal, bit 1 is known as the lock retry signal. Explanation of these signals first requires an explanation of how reads work on BLT.

In a normal read a CPU sends out the address and owns the on card bus until the data is returned. BLT does not naturally use this model. BLT attempts to complete transfers to a card while the card is waiting for the read data to come back. This implies that the CPU must start the read, then give away the on card bus. BLT will conduct several transfers and at some point return a read response. The CPU is given the bus back, and fed the data it requested. This operation is called a split read.

If a card can do a split read, everything is great. The read retry pin should be asserted low on power up. If a card cannot do a split read, the read retry pin should be asserted high. This indicates that in the case of a read collision, a retry must be generated. If the card's read loses the internal arbitration a read response retry packet is returned. Instead of generating a normal response to the CPU, the transaction should be retried. The point of this is to allow BLT access to the card. A retry conventionally implies an arbitration of the processor bus. BLT should win this arbitration and conduct the necessary transactions. When it is done the bus is returned to the CPU which should attempt to complete the read again.

The lock retry signal indicates the identical operation with respect to locks. BLT normally attempts to conduct transactions to a card that is doing a remote lock. If a card does not support

this, it should assert the lock retry signal. This will cause BLT to generate a retry packet in the case of colliding locks. Note that read retry always implies lock retry, since a card that retries on locks must initiate a locked transaction with a read.

The low order three bits of packet type are used to define the interface mode. The encodings are as described below. If an illegal mode is indicated BLT will not function and will remain tri-stated. The system will not realize that the card exists. All transactions to the card will generate address errors.

Table 3-2
Card Configuration Bits

011	8 bit separate interface
010	16 bit separate interface
001	32 bit separate interface
101	32 bit multiplexed interface
100	64 bit multiplexed interface

The implementation of system initialization is done through the serial bus. After the CPU has come up it accesses registers internal to BLT. These reflect the values on the configuration lines. The CPU configures the crossbar and then brings the card out of reset. The system proceeds by loading the card firmware and doing any card specific initialization.

The card firmware is provided by a ROM almost identical to Nubus. Some of the configuration details have been changed, but overall the concept and structure of the ROM is identical. This is discussed more completely in the software section.

Interface Modes

As mentioned before, there are five different interface models for BLT. A card is free to pick whichever one is most convenient. The mode a card is using is defined during card initialization and cannot be changed after that.

This section goes into details on the timings of the various modes. In order to save a few trees, all five modes are not individually drawn. An example mode from multiplexed and non-multiplexed is used. The timings for the other modes are exactly the same as for these example modes. The only differences being the number of significant bits in the twist and size lines.

The diagrams are simplified as much as possible. Signals that do not change are left out. For instance, on slave writes none of the output enables are drawn. All are assumed asserted.

In all timing diagrams a type face convention is followed. If the signal is in normal type it is driven by BLT to the card (an input) for the entire diagram. If a signal name is underlined it is driven

by the card to BLT (an output) for the duration shown. If a signal is in italics it is driven by both devices at some point during the diagram.

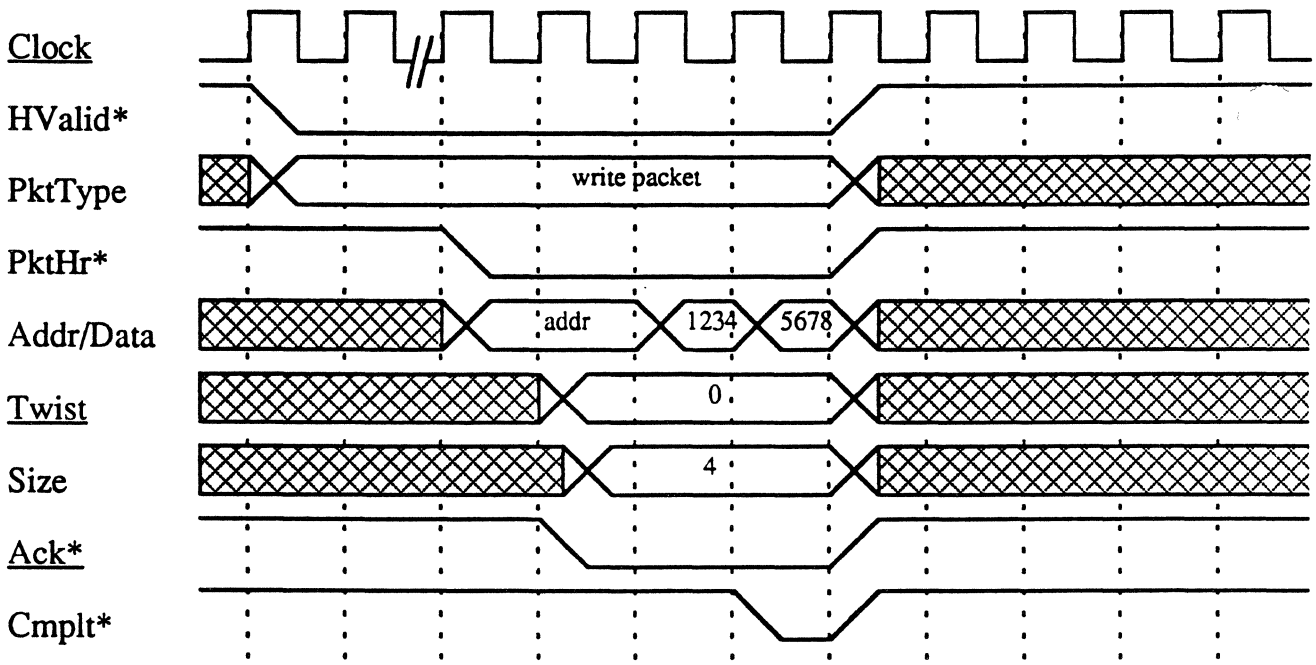
No timing information is implied by the diagrams, simply transitions based upon edges. The exact timing for setup and clock to Q will eventually be specified.

Multiplexed Modes

There are two multiplexed interface modes. Both share identical timings, so only one will be discussed in detail here. Because it makes the diagrams smaller, the 32 bit interface is discussed here. The 64 bit mode differs only in that the upper bit of the twist and size lines are used.

The four fundamental operations are discussed in this section: slave read and write, and master read and write. All will be done around a 8 byte packet to an aligned address. This hides some of the complexities that can make diagram drawing a nightmare. A separate section discusses the twist and size timings in detail.

Figure 3-3
32 bit Multiplexed Mode Slave Write, 64 bit Packet



This is a generic multiplexed transfer. As with all slave transfers it begins with BLT asserting header valid. At that time the packet information becomes valid. For simplicity, the header output enable is not shown, but is presumed asserted. This is only an informational signal, no action by the card is necessary.

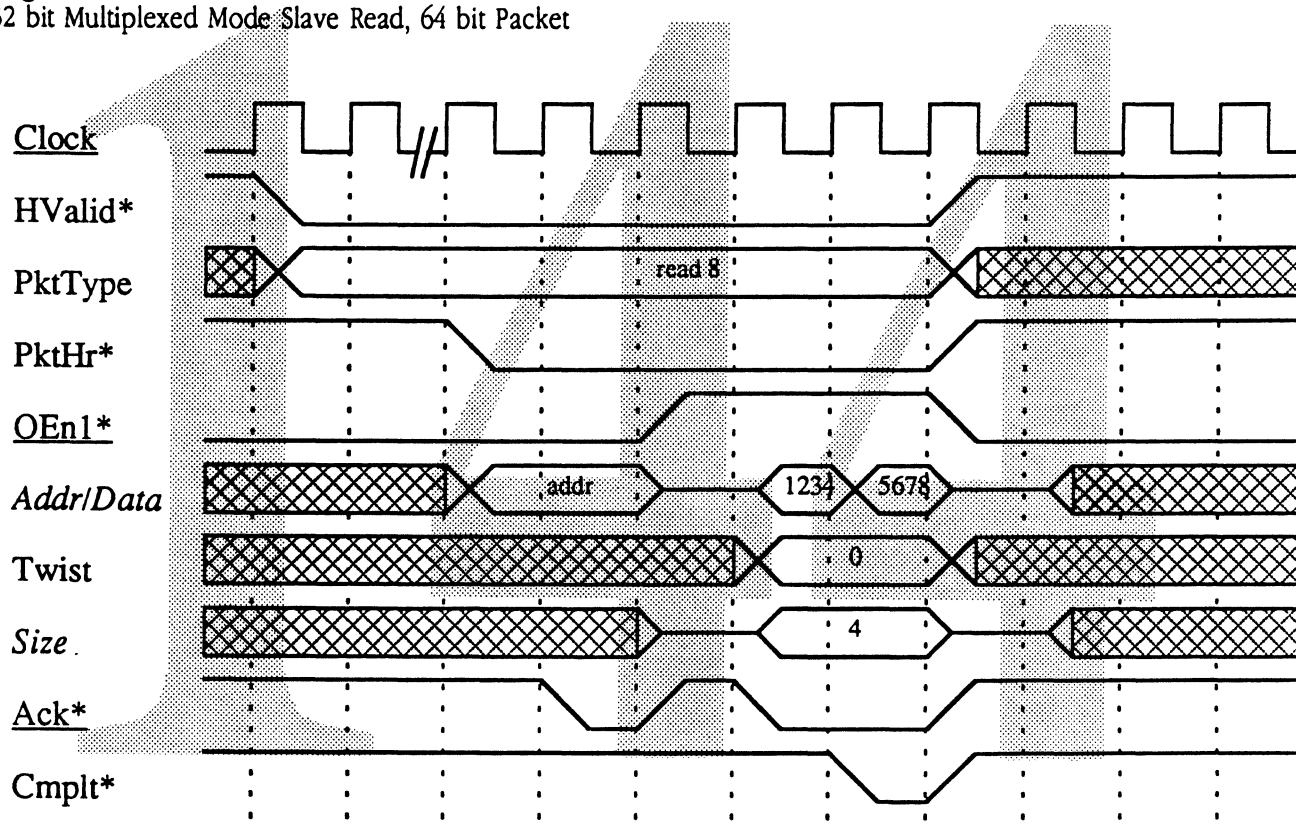
At some time later, a delay that will vary between BLT implementations, the packet here signal is asserted. This indicates that the address is valid so actual data transfer can begin. In this diagram the card begins doing single tick transfers by asserting the acknowledge signal and holding it

low. Thus, two ticks later the 64 bits have been transferred. Note that it is possible to slow the data transfer by simply not asserting acknowledge.

The complete signal becomes asserted on the last tick. On the next rising edge with acknowledge asserted the cycle will be complete.

All signals in this diagram are synchronous to the rising edge of the bus clock.

Figure 3-4
32 bit Multiplexed Mode Slave Read, 64 bit Packet



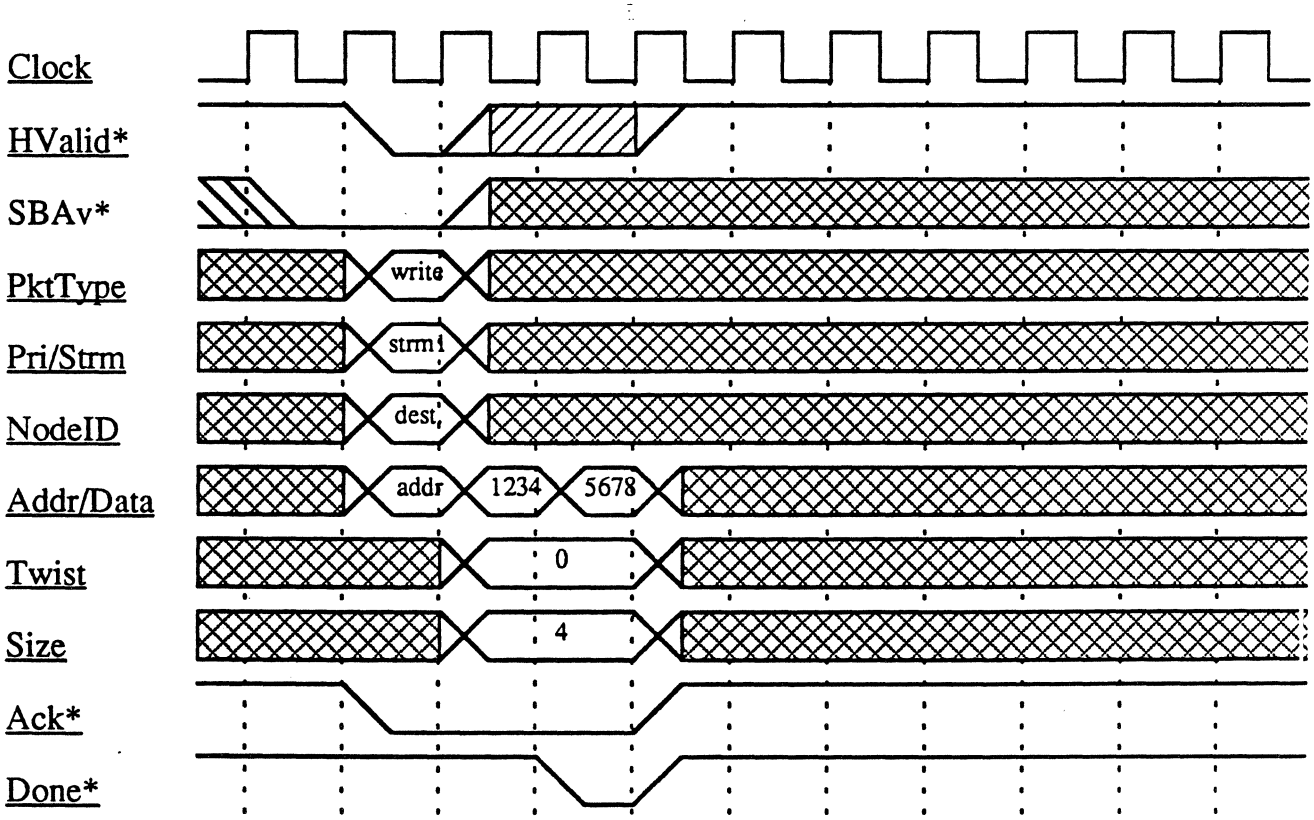
This diagram shows a multiplexed slave read. The most interesting part of this diagram is the change in direction on the data bus. When the packet arrives, the data bus is driven by BLT because the output enable is asserted. The card acknowledges the address, creating the response packet. The card then reverses the direction of the data bus. This is done by deasserting the output enable signal. Note that BLT tri-states immediately upon deassertion of this signal. However, data cannot be driven into BLT until the subsequent tick. This is done to guarantee a dead tick to prevent bus contention. BLT treats the output enable as synchronous for its assertion so in the diagram when it is asserted, data is not driven by BLT until the next tick as is shown by the data and twist lines..

The size bits share direction with the data bus. In this example the size bits are driven by the card. It is presumed that the card is driving the output enable zero line with the same value as the output enable one line as suggested in the signal description.

The rest of the cycle precedes normally. The two words of data are loaded into BLT by assertion of acknowledge. On the last tick, the complete bit is asserted, indicating that the last data

is about to be loaded. This allows a simple card to avoid having to count how much data it must return. It should simply load data until it loads a word when BLT has asserted the complete signal.

Figure 3-5
32 bit Multiplexed Mode Master Write, 64 bit Packet



As can be easily seen from this diagram, BLT is great for writes. The entire transfer of 64 bits takes just three ticks. The cycle begins with the card asserting header valid. This creates the new packet with the header information as defined by the card. At the same time, the address is loaded by asserting the acknowledge signal. The next two ticks load data because the acknowledge signal is held asserted. Again, this could be slowed arbitrarily by the card deasserting acknowledge.

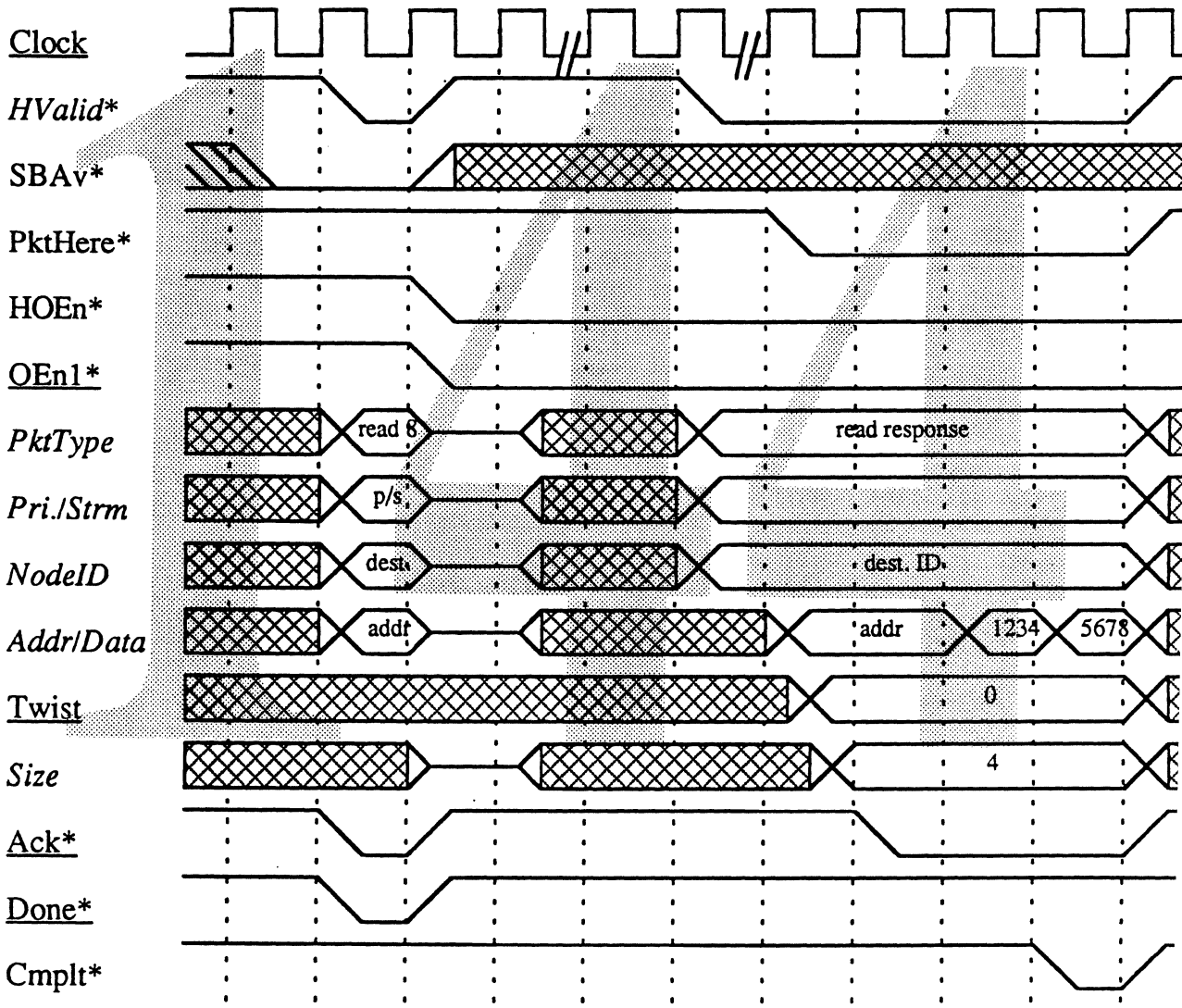
During the final acknowledge, the done signal is asserted indicating that the entire packet has been loaded. BLT then transfers the packet and makes sure everything is handled correctly for the slave device.

There are two interesting features of this diagram. The first is the header valid signal. The first tick that header valid is asserted along with send buffer available a new packet is created. This packet is the only master packet that can exist until the transfer has been completed. For this reason, the header valid signal is ignored until the packet is completed by the done signal.

The other interesting item about this transaction is send buffer available. If the creation of this packet consumes the last available buffer, send buffer available will transition deasserted on the tick after header valid is sampled. If this packet does not fill the buffers, send buffer available will

remain low. A packet is only created when header valid and send buffer available are both asserted (the first by the card, the second by BLT). The promotion section which follows later discusses what happens if header valid is asserted by the card when send buffer available is deasserted.

Figure 3-6
32 bit Multiplexed Mode Slave Read, 64 bit Packet



This diagram actually contains the timing for two separate ticks. The first is a master read. This cycle begins as all master cycles with the card asserting header valid along with the necessary header values. In this example the address for the read is asserted simultaneously. Since that is all the data in the packet, the done signal is also asserted. This launches the read in a single tick.

Note that a master read follows the same rules as master writes with regards to the send buffer available signal. A master read consumes a packet just as all other master transactions. This means that it must have a buffer before it can be launched.

The other cycle diagramed here is a read response cycle. This cycle is essentially identical to a slave read cycle. The only difference is the packet type is read response instead of write. The cycle begins with BLT asserting header valid (note that all busses have been turned around from the launch of the read packet). When the packet is ready the packet here signal is asserted. The card can then receive the packet exactly as for a slave write. On the final clock tick of the response packet complete is asserted by BLT.

The breaking of a read into these two cycles has many effects. The most important is that it allows the master device to launch other transfers, including reads. If a master has multiple reads outstanding (that is, it has sent more than one read packet without receiving a read response) it must decipher which read a response is related to. Just as for write operations, only minimal order is defined for when a read will be responded to. Two reads to different nodes will often complete out of order. Similarly for two reads of different priority levels.

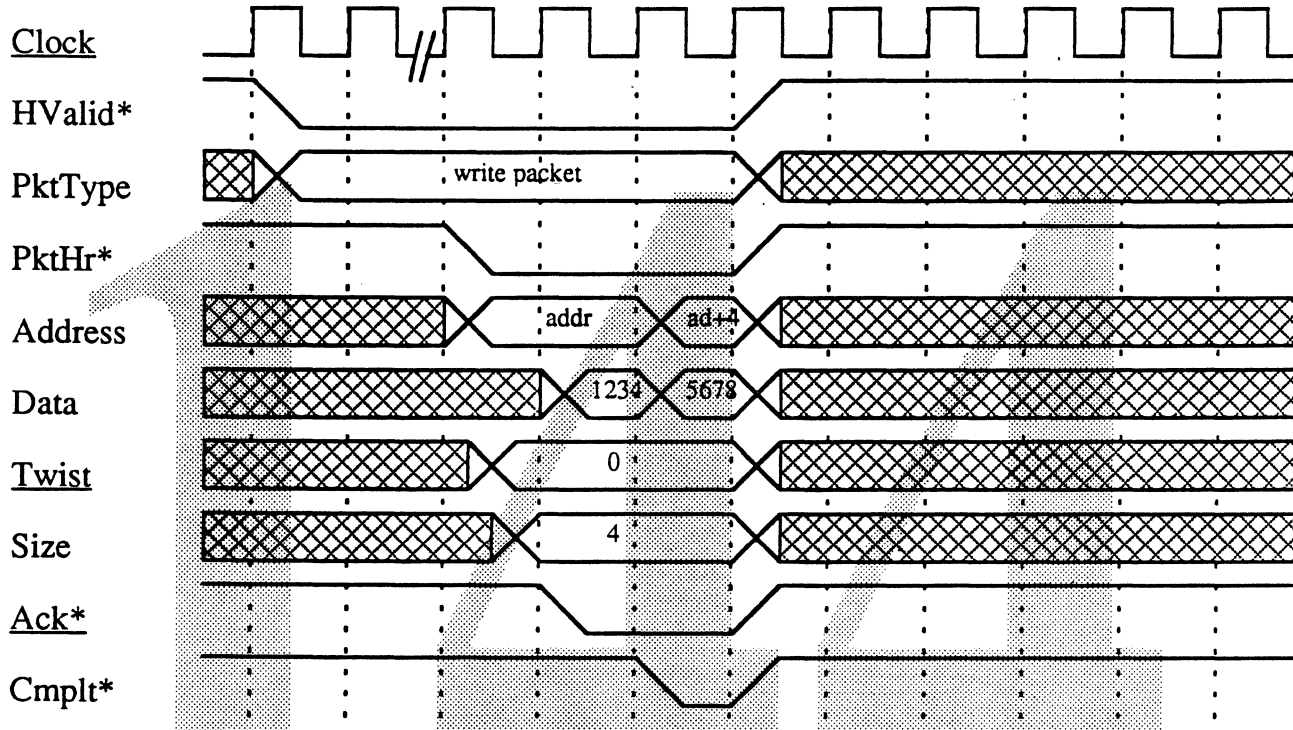
The way to decipher which response is related to which is by use of the node ID and priority bits. These bits are guaranteed to be equivalent to the ones on the original read packet. This combined with the order defined with priority and destination is guaranteed to be sufficient information. Not the most convenient mechanism, but for cards that can launch multiple reads the complexity is not a great burden.

Separate Modes

There are three distinct separate configurations. As in the multiplexed modes description only a single one is documented here. The only difference between the modes is bus width and the size of the twist and size lines. In 8 bit mode the twist and size lines have no effect and can be ignored.

All the timing diagrams are for a 32 bit interface, transferring a 64 bit packet to an aligned address. The complexities introduced by the twist and size bits are discussed in a separate section.

Figure 3-7
32 bit Separate Mode Slave Write, 64 bit Packet

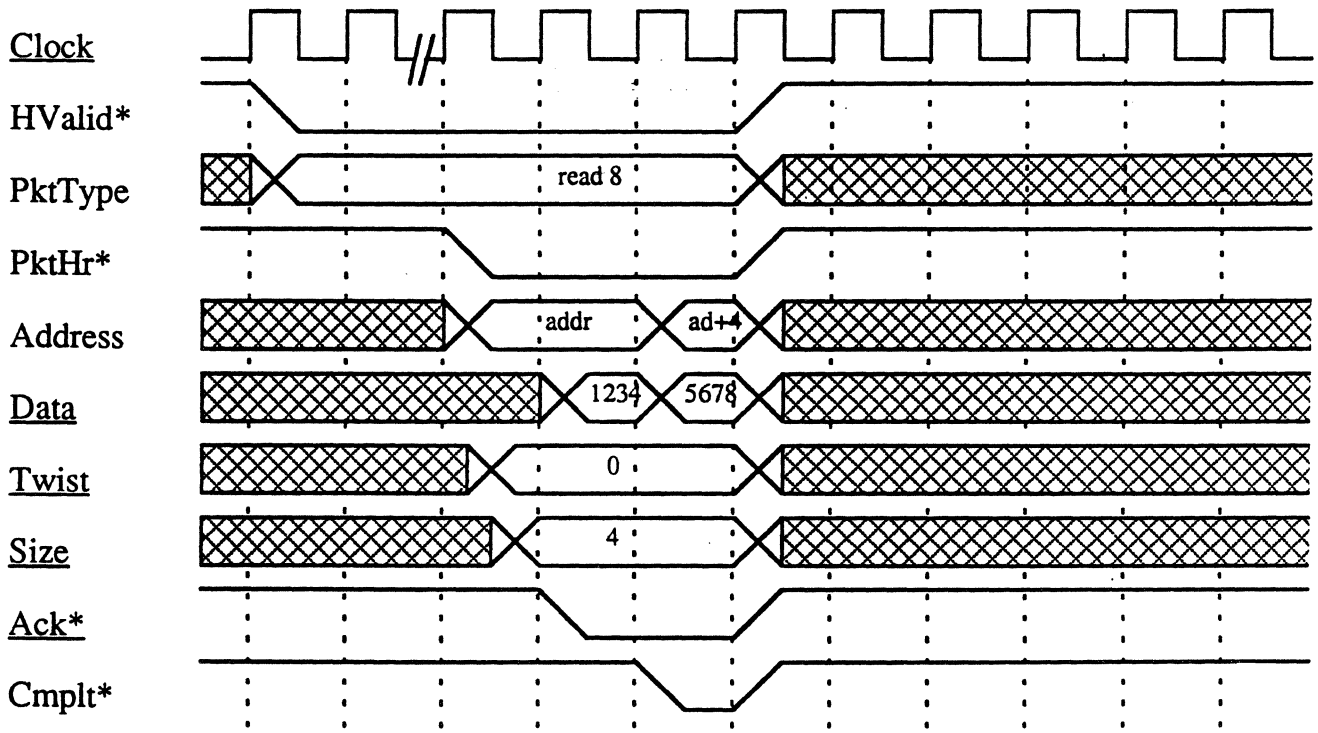


The above diagram is about as generic as a transfer on BLT gets. As always for slave transactions it begins with header valid being asserted. Packet here signals that the address and the first word of data is available.

The cycle differs from multiplexed when the card asserts acknowledge. In separate slave interfaces the address is kept consistent with the current data word. When the card acknowledges a word the address is incremented by size. In this case 4 bytes are transferred on the first word, so the next tick address is incremented by four.

The complete signal is asserted for the last word transfered just as in the multiplexed modes. Upon acknowledge of this word the header valid and packet here line are deasserted for this transfer. They both could remain asserted, indicating another pending transfer.

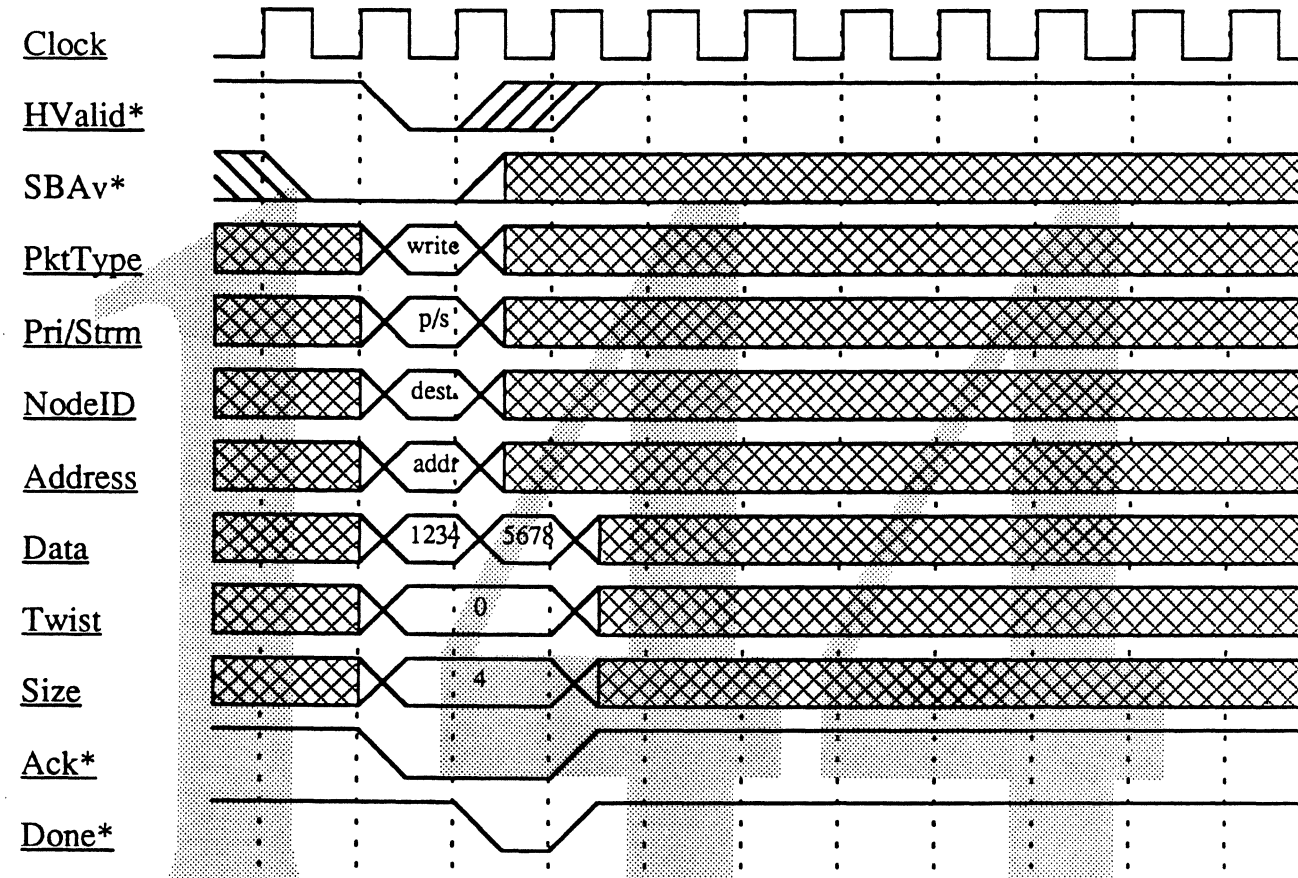
Figure 3-8
32 bit Separate Mode Slave Read, 64 bit Packet



This diagram shows a standard separate slave read. It begins exactly like any other slave cycle. The first acknowledge indicates that the first word of data is being asserted by the card. As in the separate slave write described above the address is kept consistent with the current word being transferred.

The complete signal performs the normal function of indicating that this is the last word to be transferred. Upon acknowledgement, packet here and header valid indicate status for the next packet.

Figure 3-9
32 bit Separate Mode Master Write, 64 bit Packet



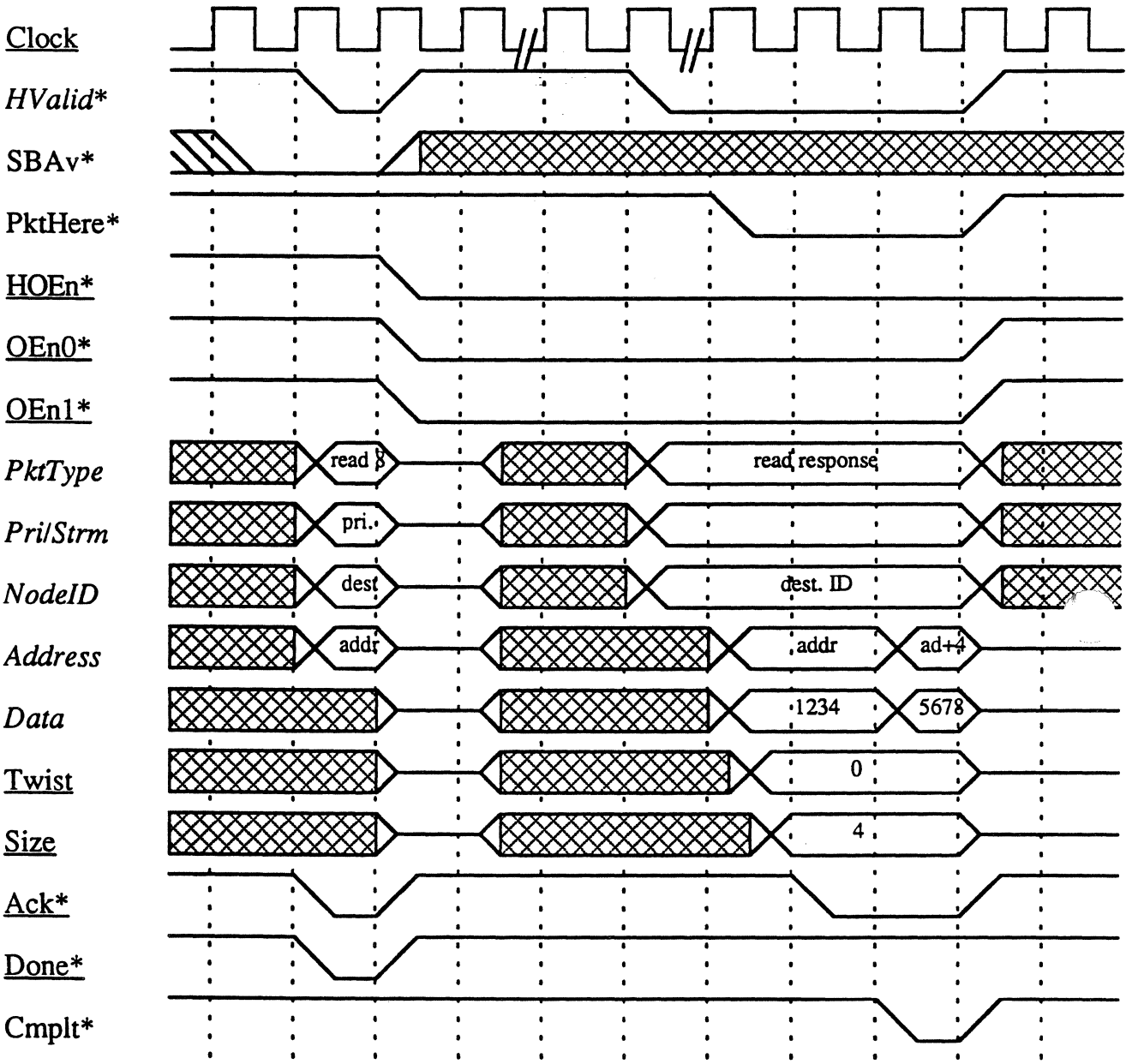
The separate write master interface shows off the efficiency of BLT. On the first tick, the packet is created by assertion of the header valid signal. Simultaneously the address and the first word of data are loaded by the assertion of acknowledge. In this single tick, 76 bits of information are passed into BLT.

The cycle completes exactly as in the multiplexed examples. The assertion of done along with acknowledge indicates to BLT that this is the final word being transferred.

The other signals in this diagram are consistent with the multiplexed models. The send buffer available indicates that a new packet can be created. The header valid signal is ignored after it has created a new packet, until the packet is completed.

An interesting point is that if the master were only transferring a single word of data, this write could be conducted in a single tick.

Figure 3-10
32 bit Separate Mode Master Read, 64 bit Packet



This diagram shows the two cycles that make up a master read transfer. The first cycle launches the read packet. This is done identically to the multiplexed mode. Assertion of header valid, acknowledge, and done simultaneously loads all the necessary information in a single tick. On a master read launch the data is meaningless, although it is loaded. Note that the size of the requested response packet is encoded into the read packet type.

The separate read response is also fairly simple. It is identical to a separate slave write. It begins with the assertion of header valid. The header information is valid at this point which allows a

card to do whatever processing is required to complete the read transfer, perhaps returning the local bus to the initiator.

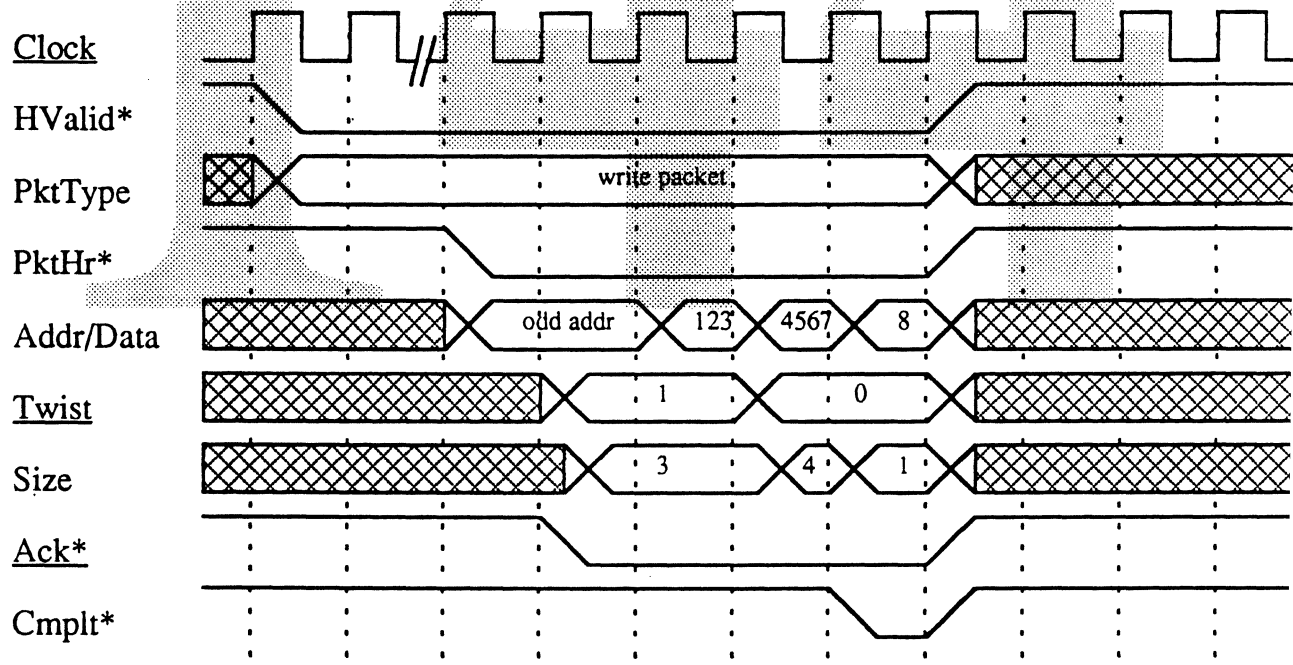
Eventually the packet becomes valid. The card acknowledges the packet by asserting the acknowledge signal for two ticks. On the last tick BLT asserts complete to indicate that the packet will be completed.

Twist and Size

The twist and size lines are one on the more unique aspects of BLT. These bits make the interface more powerful as well as easier to use.

The basic function of the size lines is to allow transfer of an arbitrary number of bytes per tick. By asserting a size of one a byte is transfered, a size of four transfers 32 bits. The twist lines allow arbitrary alignment of this data. To transfer a single byte on the third byte lane, the twist lines are asserted to 5 and the size lines to 1 (see the figure 3-2 for an explanation). Together they provide most of the functionality traditionally provided by byte enable lines.

Figure 3-11
32 bit Multiplexed Slave Write, 64 bit Packet, Misaligned address



This diagram shows perhaps the most common usage of the twist lines, a non-aligned transfer. In this example the address has the low order bits of 01, which the card is latching and then driving to the twist lines.

On the first data tick the twist lines are set to 01. In response, BLT adjusts the size bits to avoid "wrapping" the bus. Since the card is 32 bits wide, this means 3 bytes can be transferred. The card acknowledges those three bytes as normal, marking the first as garbage.

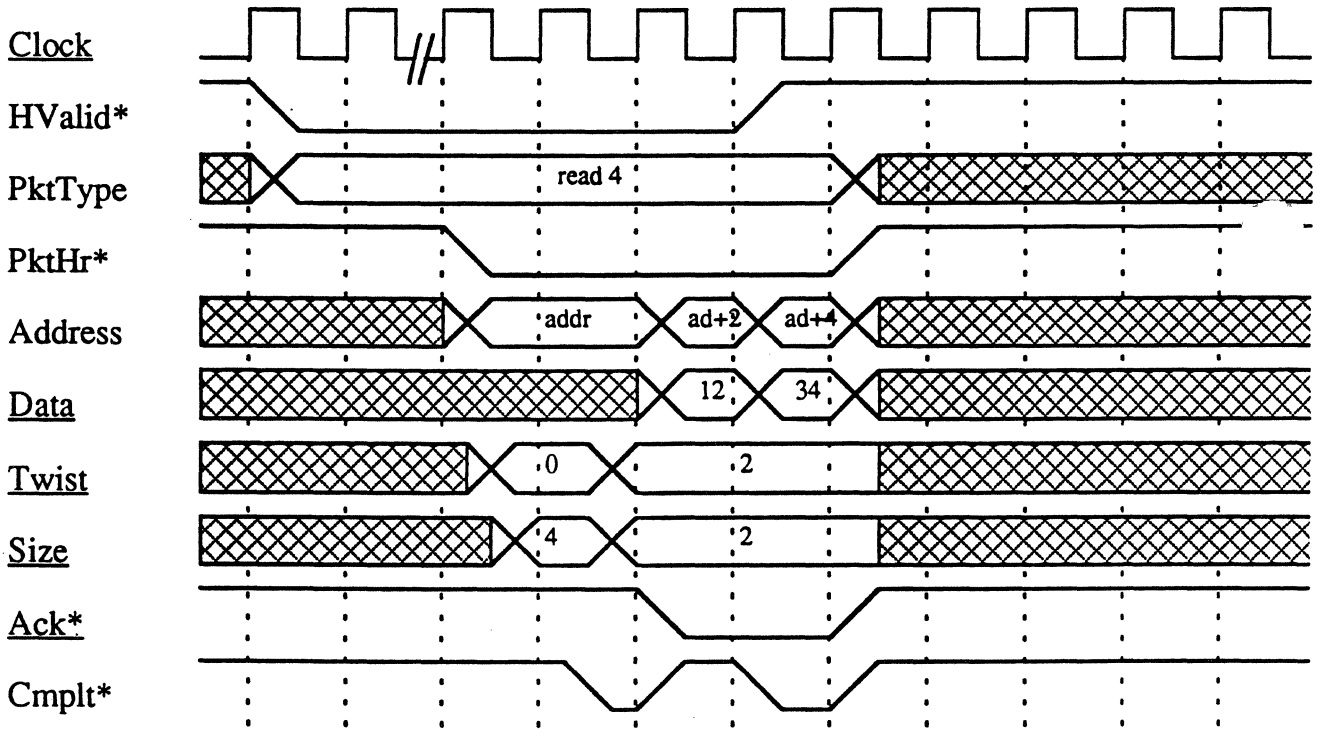
At the conclusion of this cycle the card clears the twist lines since the address is now aligned. BLT responds by asserting the size lines to 4, a maximum size transfer on a 32 bit bus. Note that the data bus contains bytes four through seven of the packet. One more byte must be transferred in order to complete the transaction.

Since the address is still aligned the card leaves the twist lines at 0. BLT synchronously changes the size lines to 1, indicating that one byte remains to be transferred. BLT also asserts complete to indicate that these are the last bytes in the transaction. When the card acknowledges this word the packet is done and the transaction completed.

This sort of transfer in a multiplexed bus requires the card to latch the address for the first tick and clear it for all subsequent accesses. In the separate modes the address lines do this exact transition so this alignment occurs automatically if the low order bits are connected to the twist lines.

Figure 3-12

32 bit Separate Mode Slave Read, 32 bit Packet, Complete Glitch



This diagram shows the bus transfer for another valuable application, dynamic bus sizing. The interface is defined at 32 bits, but some device on that bus is only 16 bits wide. In a normal bus this would either require special software to deal with the missing bytes or special hardware to deal with twisting the data around. In BLT the hardware is built in.

The transfer begins with the bus asserting header valid and then later packet here. The card decodes the address and realizes that it is to a reduced bus device. Accordingly is changes the twist (and size) lines to reflect this. The twist lines start at 0, the address asserted by BLT but are overridden to reduce the bus size to 16 bits.

After the card has set up the twist and size line appropriately it conducts the transaction as normal. BLT transfers only two bytes per tick, aligned to the proper byte lane. Note that it also keeps that address bus current with the transfers despite the size change.

The complete bit shows an interesting glitch. While the card is asserting size 4, BLT asserts complete because there are only 4 bytes in the packet. However, when the card changes the size bits to 2 complete also changes. Note how complete changes in the midst of a clock tick. It is combinatorial with respect to the twist and size lines. It is the responsibility of the card to guarantee that complete has settled before asserting acknowledge. This involves simply waiting the appropriate amount of time.

For slave reads, is possible to play this trick for any byte lane. The small device could be on any byte lane. This is because for reads the card drives the size bits. In order to allow BLT to write a reduced size device, it must be on the highest order byte lanes. This takes advantage of BLT never wrapping the bus. The above example of aligning a 16 bit packet on byte lanes 2 and 3 allows both reading and writing via BLT without any difficulty.

Special Transactions

The transactions in this section are strange in some way. Most of them share protocol with the conventional cycles above. They are separated out as much for explanation as for specific protocol inconsistencies. Most are probably rarely used, but there existence is required to support a computer of the nineties.

Locked Operations

The locked operations on BLT have been specially designed to make implementation with todays CPUs easy. They are not the method of choice for doing atomic transactions in a packet switch environment.

A locked operation is initiated by a master card that loads a packet when the lock bit asserted. Once this first transfer has been completed, the master is guaranteed exclusive access to a single card. It is important that the master does not do any transactions to other cards while it has a card locked as this can deadlock the bus. The lock on the remote card is maintained until the master deasserts the lock bit. This deassertion does not have to be associated with any specific transfer, it can just go away. If it is brought low during a packet load that packet will be transferred before the remote card is released. The lock signal description describes the exact interaction in detail.

Depending upon the lock abort configuration, BLT handles collision in locked operation very differently. If the lock abort signal is not asserted, BLT expects to be able to conduct local transactions while this card is conducting a locked operation over the bus. It will even attempt to conduct locked transactions to this device. This does not violate the atomic nature of a lock. This simply defines that the lock is a resource lock that guarantees exclusive access to some resource, but has no effect on any other resource. The lock of the remote card does not imply a local lock.

If the lock abort signal is asserted BLT assumes that if this card is conducting a locked operation on a remote card the local card is also locked. This can cause some difficult collisions if two cards attempt simultaneous locks. BLT will assert the abort signal to the card that loses the internal arbitration. Unfortunately this arbitration can only occur once the packet is complete. If the

packet is a write this is after the writing device thinks it completed the transfer, so the abort signal will not have the desired effect. To correct this, the first cycle of a locked transaction must be a read. This guarantees that it can be correctly aborted if necessary. This restriction only applies to devices using the lock abort protocol, which includes the Jaguar motherboard.

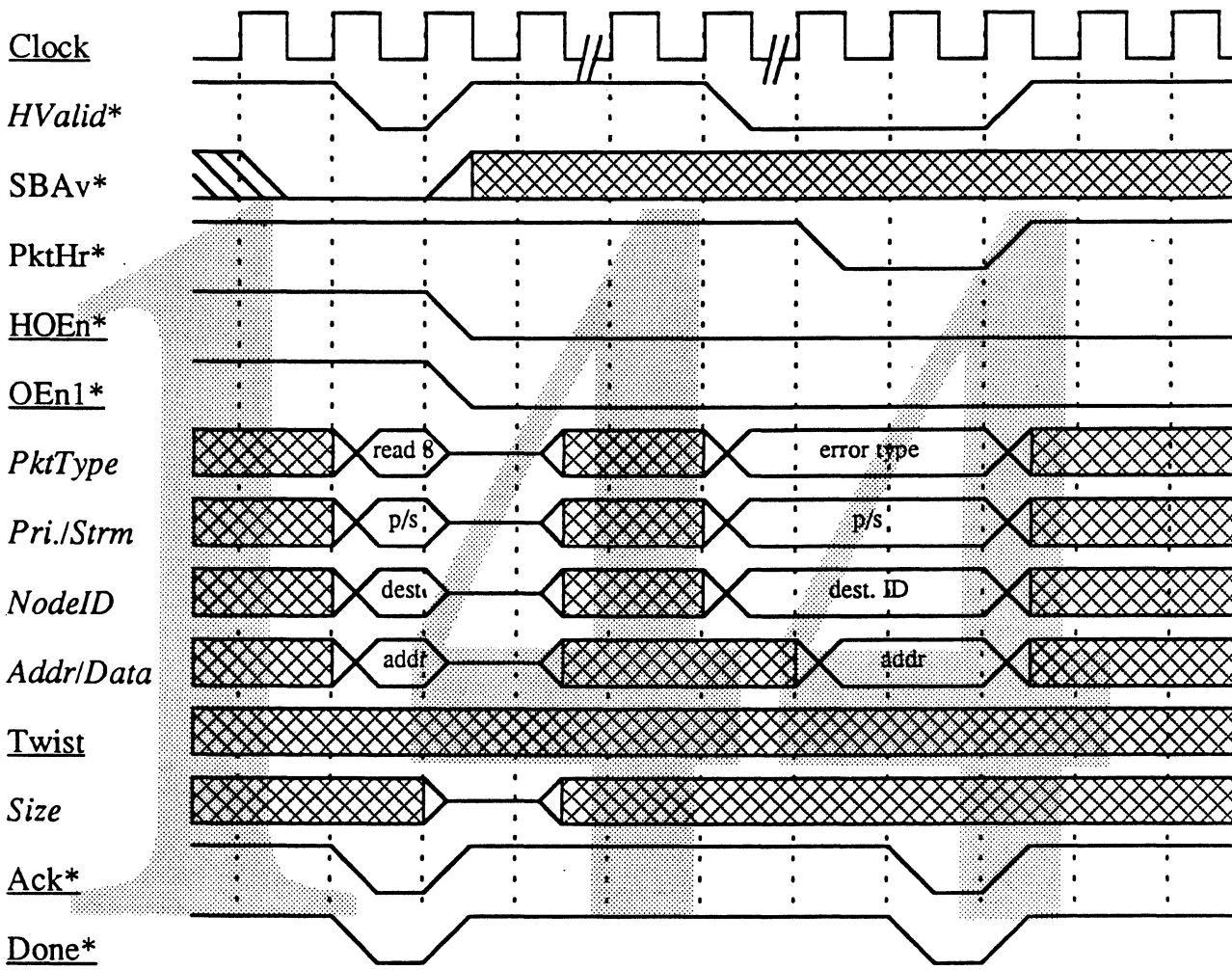
Read Errors and Retries

On BLT read errors are dealt with in a single manner. Any time a read causes an error, a header is returned with a packet type that identifies why the error occured. The most common of these is a retry. This indicates a bus collision that had to be resolved by removing one of the read operations. This should be handled by giving BLT access to the local bus, and then attempting the read again.

The other three error response packets indicate more significant errors. These range from an address error, in which a card does not exist, to a timeout which indicates that the card exists bus did not respond.

All of these response packets are identical in format. They contain no data, consisting only of a header. They otherwise appear identical to a normal response. The node ID and priority/stream information is set to the value of the original header.

Figure 3-13
32 bit Multiplexed Master Read, Error Response



This diagram shows a normal master read operation. However, the return is an error packet. An error packet is presented to a card exactly like any other packet. First the header valid signal is asserted. Sometime later the packet here signal becomes active. The address presented is the address of the read. The packet is completed by the assertion of the acknowledge and the done signal. The timing of this cycle is identical to the timing of the slave split read diagramed below.

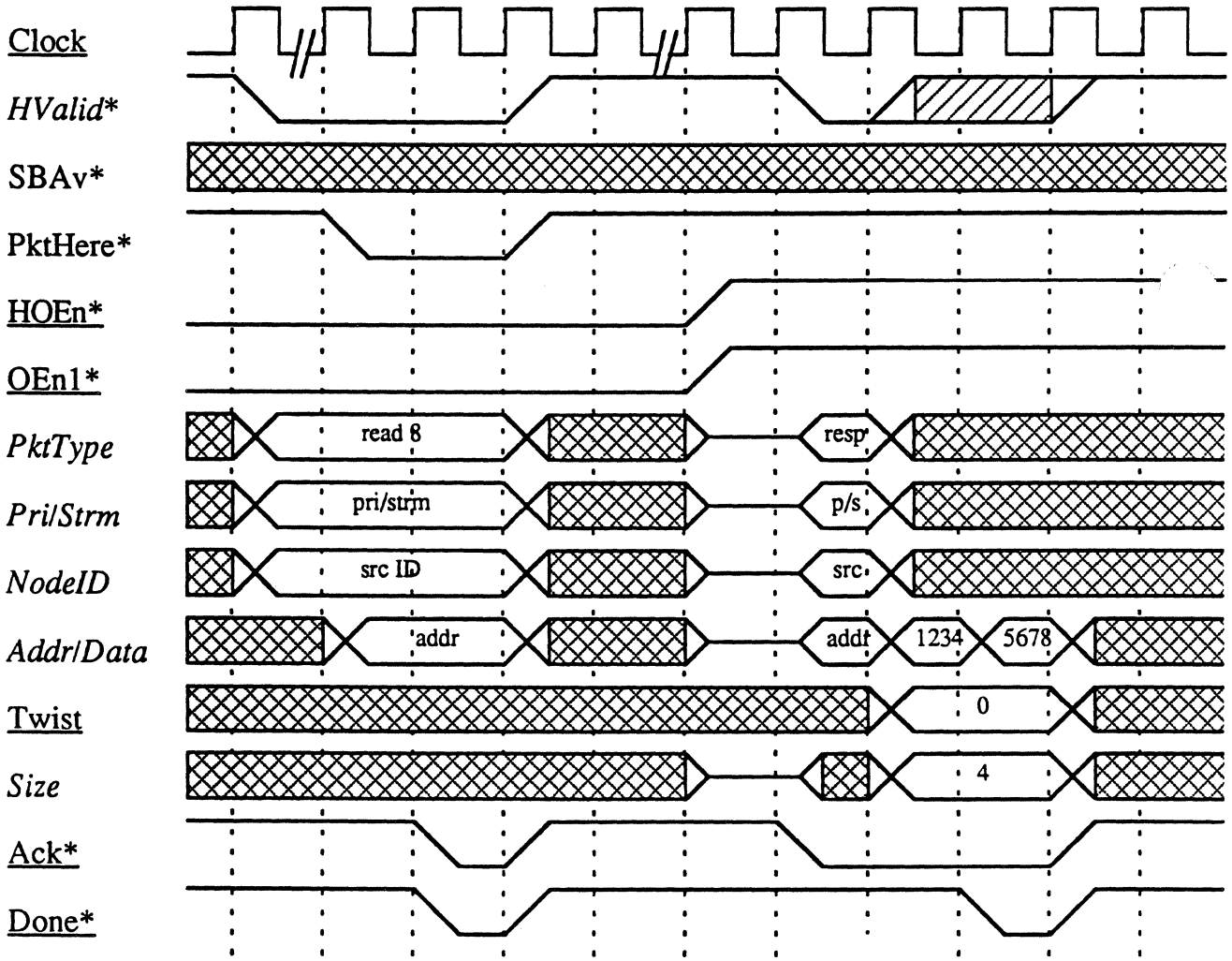
Slave Split Read Response

A normal read response on BLT is fairly simple. The read arrives at the card, the card fetches the data and puts the data into BLT. When all the data is back BLT terminates the read. This is very convenient for simple slave cards. It is not optimal for high performance processor devices like the Jaguar mother board.

Just as BLT expects to be able to complete transfers while a card is waiting for a read, the card can expect the same from BLT. A card can send a new packet anytime it is not actively dealing with a packet. If a card has not acknowledged any section of a packet it may initiate a different transfer. For instance a card may receive a read from BLT. It may look at the header and address, decide that it doesn't want to deal with it at this time, and so reverse the header direction, create a new packet and send it off. This is the normal level of split transactions.

Beyond this it is often useful to service more BLT transfers while fetching data for a BLT read. In this case the read packet arrives, the address and data are looked at, and then done and acknowledge are asserted simultaneously. This disposes of the read packet without passing back any data. This frees the BLT interface to deal with the next packet.

Figure 3-14
Slave Split Read Transaction



This diagram shows a complete slave split response transaction. It begins as a normal slave read operation. Header valid is asserted, followed by packet here. Now the transaction diverges

from a normal read. Instead of simply asserting acknowledge in response to the packet, the card asserts done and acknowledge. This "completes" the read packet. BLT deasserts header valid and address and throws away all related state, exactly as for any other completed packet. It is now the cards responsibility to create a read response packet.

The second cycle of this diagram is the split read response cycle. This begins as any master write with the assertion of header valid. In this special case, the send buffer available signal is ignored because BLT guarantees that a buffer is available to complete this. The header asserted is entirely defined. The packet type must be read response. The priority bits must be the same as for the read that caused the transfer. The node ID must be the card that originated the read.

From that point on, the cycle is a typical master write. The master loads the requested words of data and asserts done during the last tick. Note that BLT provides no support in terms of counting the number of bytes to transfer.

BLT will guarantee that if a read is shown to a card, a buffer exists for that read to be stored in, independent of the send buffer available. This means that a split read response will always generate a packet, regardless of the send buffer available status. This model is required in order to avoid a deadlock situation.

There are some very dangerous aspects to slave split reads. BLT depends upon the cards to complete the reads in a sensible order. Sensible is defined such that accesses from the same node and at the same priority are completed in the order BLT presents them. BLT maintains very little state with regards to slave split responses. It is not able to avoid launching two reads that have some order dependency.

A card must be fairly complex in order to gain any advantage from using split responses. In those few cases (such as the Jaguar motherboard) where they are applicable they can drastically improve performance. They are implemented as simply as possible by BLT. A large chunk of the complexity is off loaded to the card.

An interesting application of split read responses is allowing an external device access to internal processor state, since the read response is now a write operation.

Priority and Promotion

Priority on BLT is a very interesting situation. It seems to be very important as a basic feature, but is not exactly clear how it will be used. The current model suggest that priority will be used to cover for the inherent inaccuracies in bandwidth scheduling. This will allow a gradation of importance between transfers that will die if they do not receive bandwidth, those that will survive, and those that merely pick up the scraps left over. Most processing falls into the last class.

All normal transactions have a priority associated with them. This governs the order in which they are presented to the destination device. The priority is absolute, with high priority transfers always passing low priority transfers. High priority transfers can starve low priority transfers, never allowing them to go. In practice this will never happen simply due to latencies internal to BLT.

Priority is defined within the header, which makes it "soft". Soft means that it can be set independently per transaction and is not linked to the node ID or other hardware feature such as a jumper.

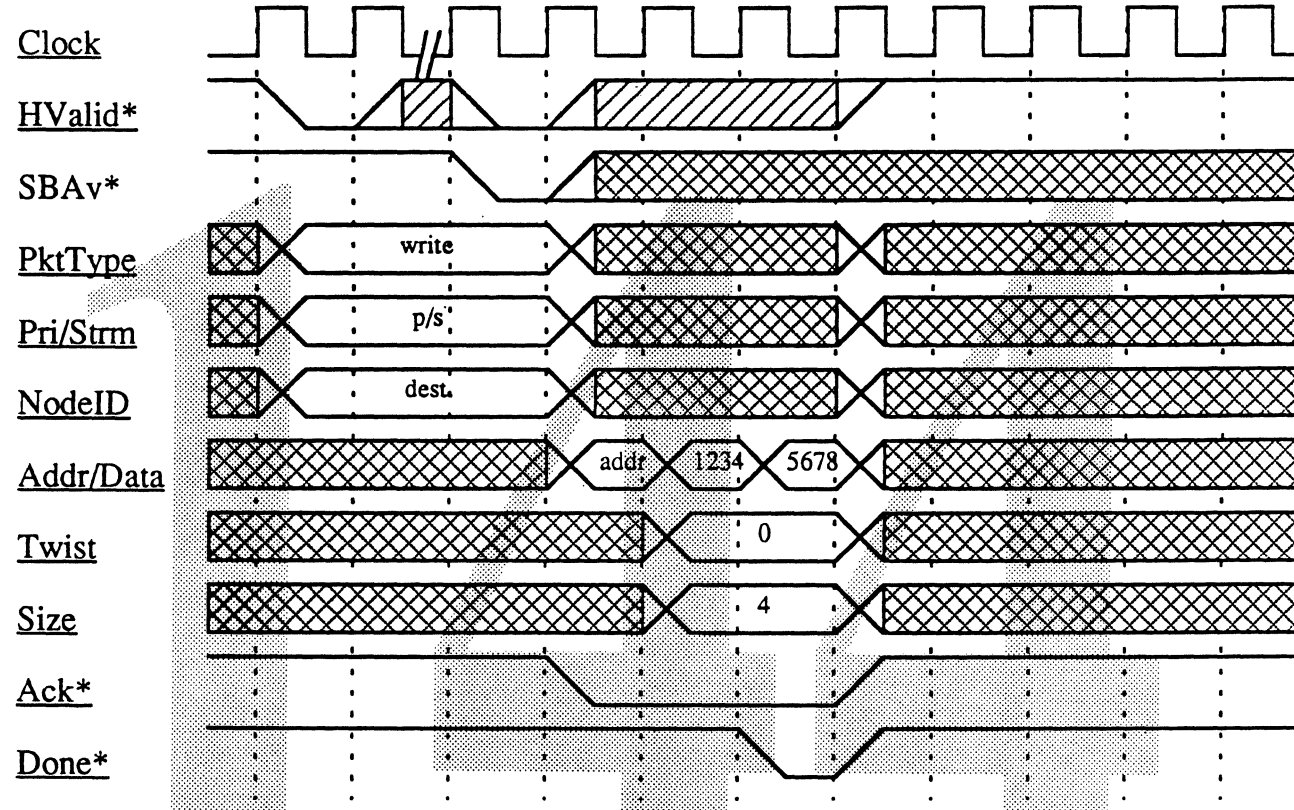
Priority on BLT is guaranteed to be fair within a priority level. If two devices are transferring packets at the same priority level, they will share the available bandwidth without any favoritism.

There are two interesting items about priority: promotion and order. To implement priority it is necessary to change the order in which transfers occur. This means that there can be no assumptions about which transaction will complete first if different priority levels are used. If the same priority level is used, then order is guaranteed. Note that the architecture of BLT fundamentally prevents there being any definition of order between transaction from or two multiple cards. This is all the responsibility of software to enforce. Locked operations make it possible for software to implement any sort of order needed.

The other interesting item about priority is promotion. This is a technique that prevent low priority packets from blocking high priority packets. Each node on BLT has a small number of internal buffers. These buffers can each hold an entire packet. On a packet load, the data is placed within one of these buffers and the header enters arbitration. Thus every packet in BLT is simultaneously evaluated for which should be serviced next. This guarantees that the highest priority packet is serviced first. Unfortunately, since there are a limited number of internal buffers it is possible that the most important packet is not being serviced because it is not currently in BLT.

Promotion avoids this problem. If all of a nodes buffers are full the send buffer available bit is deasserted. This indicates that no new packets can be created. If the header valid signal is asserted anyway, a promotion is performed if necessary. If the priority on the incoming header is higher than any currently stored in BLT, one of the internal packets is promoted. The packet selected is the oldest, highest priority packet. The priority of this packet is changed to the priority of the incoming packet. This makes the external packet arbitrate correctly with all internal packets.

Figure 3-15
Packet Promotion



This diagram shows a typical packet promotion example. The master desires to begin a high priority write but send buffer available is high. The card asserts header valid along with the header values. Internally BLT promotes the appropriate packet. Nothing else occurs when header valid is asserted. No data is loaded and no packet is created. The card must reassert header valid when send buffer available becomes asserted (it is possible to just hold header valid asserted, guaranteeing that on the first tick that send buffer available is asserted a packet is created).

When send buffer available becomes asserted a normal transaction can occurs. The promotion has absolutely no effect on any subsequent transaction. A promotion does not even have to be followed by an actual packet.

Note that since read responses use a separate internal buffer they are never promoted. This makes it possible to link read responses to reads via priority if multiple reads are launched. Stream packets will also never be promoted. With proper design a card sourcing streams should track its internal buffer carefully enough that streams will never block priority transfers.

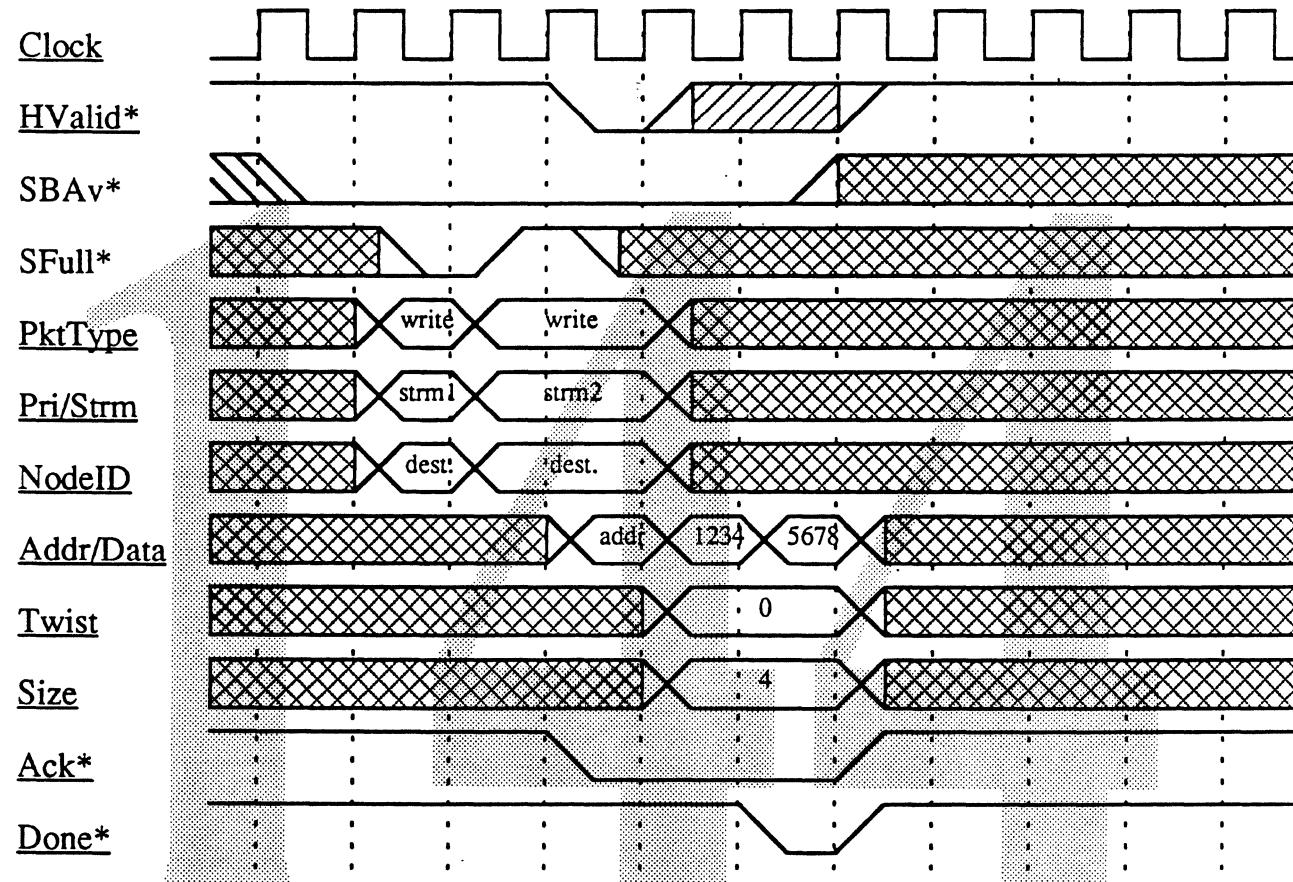
Streams

As mentioned in the introduction, streams are used to implement flow controlled writes. This allows data transfers to take advantage of the highest performance mode of BLT without sacrificing the ability to conveniently block the flow of data.

There are two parts to streams, sending them and and receiving them. Sending a stream is identical to sending any other packet. The only difference is in the values asserted on the priority/stream control signals. Any card that can write a packet is also capable of writing a stream simply by changing the data in the header. This functionality is fine for a device sourcing only a single stream. If the destination chooses to block, the packets will build up in the internal buffers. Eventually all the buffers will be full and the card will not be able to send more data; a very effective means of flow control.

This is fine unless a card is sourcing two or more streams. Streams are almost always used for data dependent operations. If one stream blocks another a deadlock could result. A mechanism must be provided for a card to monitor the progress of individual streams. The stream full bit described in the signal section is exactly that. By asserting the stream number about to be transferred a card can query if there is a packet already in BLT. If so the new packet is not loaded, leaving buffers for other transfers.

Figure 3-16
Stream Status Query



This diagram shows a typical stream status query. It is done by simply asserting the header values for the stream in question on the header lines. The stream full bit is combinatorial from these signals, so it is the cards responsibility to wait enough time to let the signal resolve. Note that this signal is always active; it always reflects the status based on the header values currently asserted.

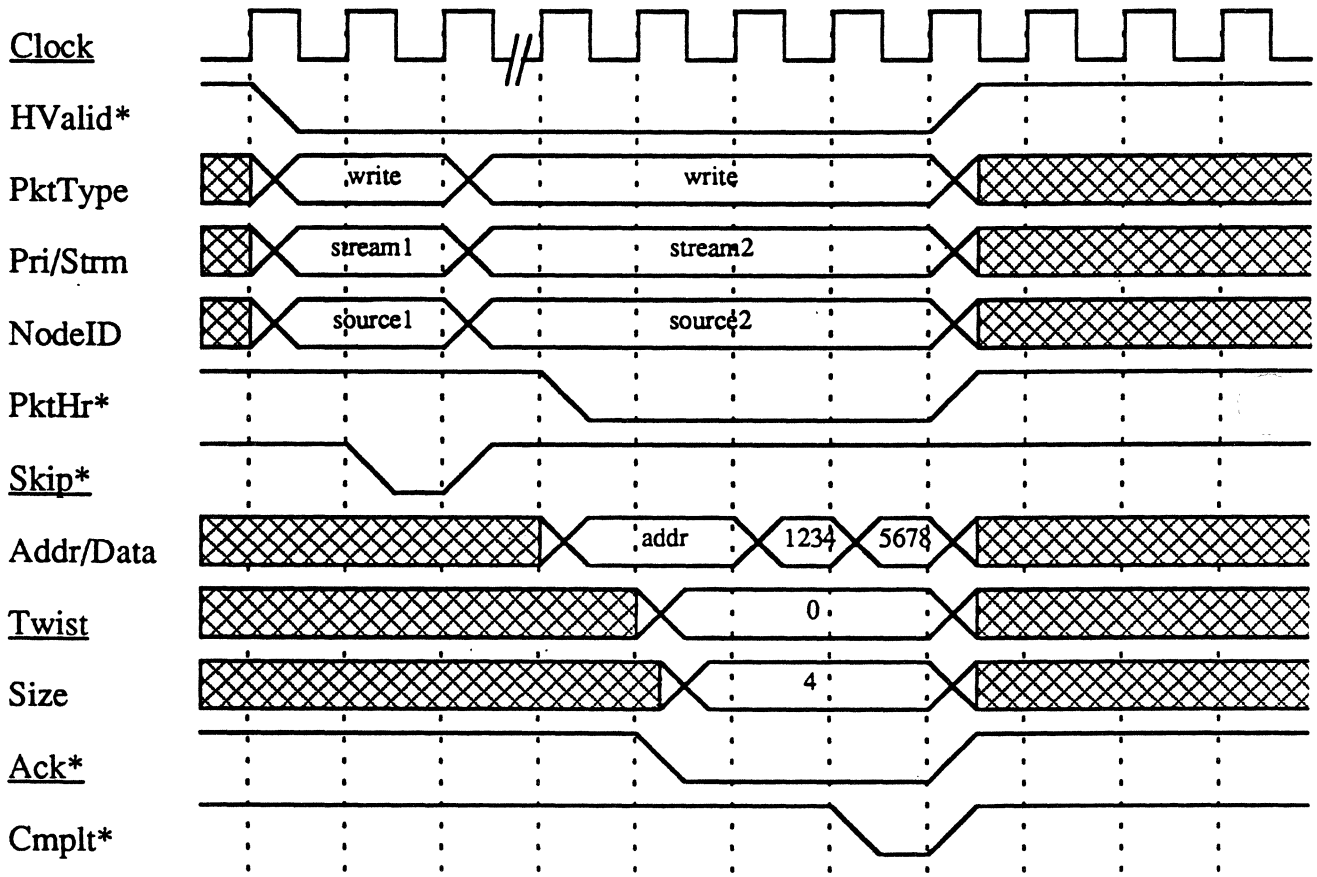
In the first tick the card asserts the header values. The stream full bit becomes asserted indicating that the packet should not be loaded. The card changes to a new stream and again checks the signal. This time stream full is deasserted. The card immediately creates and transfers a new packet. The packet transfer itself is identical to a normal master read, in particular the send buffer available signal must be monitored.

With correct use of the stream full signal up to the four simultaneous data dependent streams can be supported from each node. This restriction comes from the number of internal buffers provided by BLT. This is the only place where the number of internal buffers in BLT becomes an issue. In reality this is only an issue for the system software that manages streams. It must not assign more cross dependent streams to a node than there are buffers within BLT. It is possible to have up to 8 streams on a single node, the number of unique stream numbers. Care must be taken to avoid cross dependencies; when service of one stream requires data from another.

BLT does not constrain what can be sent as a stream. It is not an error to send a read packet as a stream. Stream packets are only special in how they react to the skip signal and that they are never promoted. Similarly, the stream full query can be used to get status for any packet except a read response.

The other part of streams is their reception. The basic piece of functionality needed is a way of sifting through multiple streams to find the one that is needed. The skip bit provides this capability.

Figure 3-17
Stream Skip



This diagram shows the use of the skip signal. BLT begins by asserting header valid to commence a slave write. The card examines the header and decides that it cannot use the packet. The card asserts the skip bit requesting the next packet. BLT presents the new packet, beginning another slave write. This time the card decides it needs the packet. It waits until packet here is asserted and does a normal write operation. The skip signal can be asserted any time until the first acknowledge has been asserted. This commits the card to servicing the packet.

The current plan is for BLT to be able to skip through headers at a rate of one per tick. However the header valid signal is the only accurate information of whether a header is actually being presented. While it is likely the current implementation will be able to do single tick skips, it is

possible that a larger BLT will have problems. Skip only has effect if the header valid signal is currently asserted.

The skip bit selects a new packet for consideration by BLT. The algorithm used is up in stream number and up in source node, starting from the last stream serviced. Thus stream 4 node 2 is followed by stream 4 node 3 followed by stream 5 node 0. The next stream presented is always the next packet in this order (if the only packets available are from stream 4 and stream 0, they will be presented sequentially). This algorithm is only specified to allow the system software to do a sane allocation of stream numbers. As implied, streams are kept independent by stream number and node number. This allows up to 8 streams per node, all of which can be fed to a single node. Order is guaranteed within a stream.

The relationship between streams and priority packets is important. Priority packets are always presented before streams. This applies after any completed transfer.

For the purposes of skip, all priority packets from a single node are treated as belonging to a single stream. If a priority packet is skipped, the next packet presented will be a priority packet from another node, if possible. If such a packet does not exist than a stream will be presented. Once all the streams have been skipped, a priority packet will be presented again. If priority packets are skipped and then a stream is serviced, a priority packet will be the next presented.

This protocol limits the usefulness of skip for priority packets. This is intentional. Skip and priority are two almost orthogonal concepts. They do not make a great deal of sense used together. The protocol is designed to maximize the efficiency of dealing with streams by jumping over all priority packets. In reality, a priority packet should rarely, if ever, be skipped.

Interrupts

Interrupts are very fundamental to a interconnect architecture. They are the basic means of getting another devices attention. As with the rest of BLT, the interrupt architecture is designed to be simple, powerful, and symmetric. Any card can interrupt any other card including the motherboard. This can be done either by asserting the interrupt line or through memory mapped interrupts.

The basic interrupt model for BLT is memory mapped. The upper 256 bytes of a cards memory are defined to be interrupt locations. A write to one of these locations will generate an interrupt to that card. These locations are overlaid on the card ROM in order to conserve memory space.

Any card that directly accesses these locations must be able to write at least the the upper sixteen locations. Thus any interrupt generated by these locations is guaranteed to be accessible by any device in the system. The rest of the location are guaranteed not be allocated for any other purpose and are accessible by most devices, so are still quite useful.

This definition means that it takes 8 bits to implement a memory mapped interrupt access device. The node ID and the four lowest bits of address must be adjustable. Forcing the rest of the bits asserted generates a interrupt location. It is important that 4 bits be allocated to node ID. All current envisioned systems could be handled with three bits, but guaranteed compatibility requires the four bits.

A card is not constrained to implement any particular number of interrupt locations. A simple card will not have any, a moderate complexity card may implement one interrupt location. The only constraint is that they must be mapped as a write operations in the upper 256 bytes of the cards address space. To guarantee generic access, they should be in the upper 16 bytes of space.

No specific value is written to the address. Multiple interrupts may be mapped to a single address which makes it difficult to guarantee that a value will actually be correct. The large number of locations is designed to alleviate this problem. Instead of relying on a value that is written, the address accessed is used to indicate which device generated the interrupt. This makes it important that a device that generates memory mapped interrupt be able to address at least the required 16 locations.

The interrupt architecture is simple and powerful, given that a card is a bus master. Many cards are not bus masters. For these cards a translation mechanism is provided through the Mazda I/O chip that maps a transition on the cards interrupt line into a memory write.

Each of the expansion cards generates an interrupt line that is received by Mazda. As for all interrupts, the Interrupt Mask register on Mazda contains a bit for each of these interrupts that controls whether the interrupt is to be passed on to XJS or not.

In addition, there is a provision for each of these interrupt lines to cause a memory-mapped write to another expansion card. This allows card-to-card communication even if the initiating card is not a bus master.

The basic idea is that hardware within Mazda is pre-conditioned so that when an interrupt line is asserted, the hardware executes a write to an address within the "destination" card. The data that is written is always zeros.

The mechanism used to accomplish this is a very simple Channel Program. It would consist of the following two Channel Commands:

```
Loop:
   Wait for interrupt, then write 0 to this address
   Branch to Loop
```

Since it is possible for there to be an active Channel Program for each of the interrupt lines, the Channel Command File in Mazda must have three locations reserved for the Wankel task that controls this operation. When the task interprets the first Channel Command above, it arms a hardware module to start looking for the appropriate interrupt line to be asserted. When it is asserted, the hardware module sends a "write 0" request to the Bus Interface Unit. The BIU reads the proper address from the Channel Command file and carries out the write operation. The task then executes the branch, looping back to the first Channel Command. The task will continue to execute the Channel Program until it is aborted by the processor.

This mechanism can best be viewed as a virtual wire. The reflection of the interrupt to a specific card does not do anything with that interrupt. The card that receives the interrupt is responsible for whatever actions are necessary to clear the interrupt line.

The latency of this virtual mapping is perhaps an issue. The fundamental latency of a Mazda task will eventually be characterized. Because of the nature of Mazda this will be fairly accurate. Of course, for maximum performance direct access to the memory mapped interrupts is available.

Errors

This is the ugly part of any bus specification, and that is a feature we have not changed.

Errors on BLT break into two basic classes. The easy ones are those that can be detected before the initiating device has completed its transaction. All read errors fall under this category. BLT is also able to detect write address errors when the packet is created. These errors can be dealt with fairly directly by software because the access that caused the error is still in progress. The normal method for reporting these errors to a processor is via a bus error.

The handling of errors on reads seems to be about optimal. The error response packet is consistent with a normal packet so it does not require special handling by the on card bus interface. Since the guilty device is still conducting an access the error can be reflected back immediately for relatively easy handling. This error is equivalent to current bus errors so the same handling methods apply.

Write errors are not quite so pretty. The handling of address errors seems correct. BLT can always assert the error signal the tick after a packet is created. In most situations this is early enough to relate it directly to the guilty transaction. This allows a system to easily mark the errant operation. This is probably not true on the Jaguar motherboard which has done several layers of dump and run writes before the transaction even reaches BLT. For the motherboard even these address errors fall into the realm of delayed errors.

Delayed write errors are the real difficulty. There are two events that can generate a delayed error. The first is a timeout. This occurs when a card that is plugged in goes out to lunch. BLT will eventually generate a timeout error. The other event that can generate a delayed error is a card asserting the error out line. This indicates that something drastic has gone wrong with the card. A card should only do this in an absolute emergency since it usually implies that the card will be disabled.

However the error was generated, the problem remains the same, doing appropriate recovery to keep the machine running. One of the interesting things about BLT is that it is theoretically possible to do full error recovery. This would be implemented if anybody could come up with a convincing reason why it should be used. As it stands now, any delayed write error will place the card in error mode. When a card is in error mode BLT will not complete any transactions to it. All attempted transfers will result in the same error as the original one. If a packet times out, then all subsequent transfers to the device will be immediately timed out.

This protocol is fairly easy for the system to deal with. A card that sends a packet which generates a delayed error will eventually be informed by the assertion of the delayed error signal. This is usually linked to an interrupt on a CPU. Eventually the CPU will respond to this interrupt and realize that it is a delayed BLT error. The CPU must use serial bus to determine which card has generated the error.

Once this is determined, the card should either be cleared or disabled. The card can be cleared by asserting the card reset line via the serial bus. The card must then be put through the normal initialization procedure. Finally the card can be reenabled for transactions. This is essentially an identical procedure as is followed for live insertion of a card.

If the card must be disabled, this can also be done over the serial bus. Simply clearing the configuration bits makes the card invisible. This will cause all subsequent accesses to generate an address error.

The system software implication of this protocol are discussed later in the software section.

Serial Bus

The serial bus is the mechanism for accessing state internal to BLT. It is used for initialization and error handling. The protocol is not defined. This means the address range is also unknown. The registers specified in this section will be mapped into the serial bus address space somewhere sensible. The serial bus itself is mapped into Jaguar address space. This has also not been defined.

Like BLT the registers are perfectly symmetric. The registers for one node are defined although four copies actually exist.

One of the interesting points on the error handling is that it is independent of who caused the error. Any device can service any error. This third party error handling should free most cards from worrying about delayed write errors.

The following interface control register (ICR) is used for card configuration and initialization.

Figure 3-18
Interface Control Register (ICR)

figure 7.1 (ICR register bit placement)

The three card configuration bits correspond to what is currently on the external configuration pins. Similarly for the retry bits. These are only valid while the card is reset and BLT is disabled. These bits are read only.

The BLT configuration bits configure the BLT interface. If these three bits are all ones, the interface is disabled and all lines are tri-stated. Setting these bit to a valid configuration value puts the BLT interface for this node into the mode. These should be set to the value returned by the configuration bits. These bits are read/write. The retry bits are identical in that they should be set to the value returned by the card configuration bits. They are also read/write.

The card reset bit controls the reset line to this specific card. If it is low, the card is being reset. This bit is read/write.

The error mode bits are also read/write. If the error mode bit is asserted high then the card has generated an error for some transaction destined for this node. Setting this bit to a zero removes the card from error mode. If this bit is a one then the error type bit is also valid. If the error type bit is a zero then a timeout was generated, if a one then a card error caused the problem. Note that these bits only become asserted for write errors. Read errors report this information back immediately and do not demand any internal state change.

The error status register (ESR) is used to access error information.

Figure 3-19
Error Status Register (ESR)

This register contains the header of the first packet that generated a write error. The valid bit indicates whether the rest of register contains meaningful information. If the valid bit is asserted then BLT will not change this register. If it is cleared then BLT will write this register with the header of a packet that is errored, and set the valid bit. Note the the ESR stores information about packets originated at this node. This is opposed to the error mode bits in the ICR register, which reflect errors that this node caused. If node A sends a packet to node B that generates an error, node A will set its ESR and node B will set its ICR.

In addition to these two registers which perform decipherable operations, there are a pair of registers that exist per node solely because of implementation issues. These two registers are identical and serve to configure the BLT data path chips. They are called the high and low datapath configuration registers (DPCRH and DPCRL).

Figure 3-20
Data Path Configuration Register (DPCR)

There is only one meaningful field in the datapath configuration register, interface mode. This should be set to the same value as the BLT configuration bits in the ICR. The only reason for these registers to exist is to inform the two datapath chips of the card mode.

Section 4

Electrical and Physical

This section will contain all the specification needed to implement a BLT board. At this point all it contains are design goals. This will be fleshed out based on product design, silicon, connector investigations, and reality.

Signal Specification

Drive

All data and address signals are targeted as driving a 50 pF load. The control signals will be somewhat lower, as they will not be connected to as many loads.

Timing

Every effort will be made to push the clock rate as far as possible. 50 Mhz is the minimum design goal. We will attempt to design for 60 Mhz or higher.

Power

The target for board power is 25 watts per card. This seems to be low, but sufficient. The exact distribution between +5, +12,-12, etc. has not been determined.

At one point high voltage power distribution was considered. This was thrown out because it did not provide sufficient savings to warrant the excess board complexity.

Physical Specification

Size

The goal is boards as large and as square as possible. Current personal computers seem designed to make layout difficult. Connector space is extremely tight on Nubus, and worse on MicroChannel, EISA, etc. Rectangular boards force extremely convoluted routing.

The optimal shape seems square, and if out of square then the long edge should be the external connector edge.

As a further argument for large boards, this is one of the few significant differences between personal computers and workstation. Current workstation have large boards that allow high functionality designs to be easily produced. This seems to be one area where personal computer should migrate in the direction of workstations.

The size of boards has a huge impact on designs. Small boards force either limited functionality at high cost due to exotic packaging. In general, the larger a board can be, the cheaper a given amount of functionality will be (with some obvious non-linearities at the low end). PC board area is much cheaper than dense ASICS, multilevel boards, or double slot cards.

The current target from the bus designers is around 8 inch square. Product design is lobbying for around 8 inch by 6 inch, which is slightly less then current Nubus cards.

Connector

The connector is currently undefined. There are many available that meet the gross electrical needs. If a through hole connector is used, the current leaning is to a 160 connector Eurocard 4 row connector. This is a longer fatter Nubus connector. This is probably the cheapest option for a two piece connector.

There is no current leaning toward any specific surface mount connector. Research continues.

The connector issue could be profoundly effected by the decision on live insertion/removal. This is a late arriving possibility so little investigation has been done into suitable connectors.

The current front runner is the Metral connector from DuPont. This is the connector specified for use in Future Bus, so volumes should be high and prices reasonable. This connector offer multiple level connections. It also comes in both surface mount and through hole.

Live Insertion

One of the goals of Jaguar is that the user will never turn the machine off. This is rather difficult to achieve without a live insertion bus. The design of BLT makes live insertion and removal quite simple.

The principle ingredient in a live insertion device is a multilevel connector. This connector guarantees that some pins make connection before and break connection after all pins at other levels. This allows sequencing of power and signal connection to guarantee that the card comes up in a known state that does not conflict with the rest of the system.

The method proposed for Jaguar requires three levels of connectors. The first level contains all of the ground pins. This provides static protection by allowing excess charge to be dissipated before the board is inserted. This is one of the most important items for safe card insertion.

The second level consists of all power lines. All other signals are on the third level. The only other special adaption is that the card must have a resistor to pull the board reset pin active when ground is applied. This guarantees that when the signals are connected the entire board is in the reset state.

The other half of live insertion is in BLT. BLT actively senses if a card is in place. If no card is connected all signals are tri-stated including the card reset. This guarantees that when the card is inserted all BLT lines are also tri-stated. It also means that the card remains in the reset state once it is plugged in, thanks to the resistor on the reset line. This allows the card configuration bits to be read by BLT, which uses them to determine that a card has been inserted.

The Jaguar motherboard occasionally scans the nodes (via the serial bus) on BLT to see if a new board has been inserted. When it finds one it configures it according to the configuration lines. The CPU then deasserts the card reset and enables the BLT interface. The card is now alive. The CPU then goes through a normal boot routine for the card, loading all the firmware drivers, and doing any necessary initialization. The card is now up and running and fully installed in the system.

The removal scheme is almost the exact inverse. An important point is that the goal is not to allow an idiot to reach into his machine at any time and rip a board out, but to allow for orderly removal of a board.

First, the user informs the system that a board should be disabled. In hardware this means disabling the BLT interface and resetting the board. The exact implications on software have yet to be determined. Once this is done, the board can be safely pulled. Again, the double level connector guarantees that the signals become disconnected before power, and the resistor guarantees that the board remains in reset state. A simple safe removal.

A scheme is also available that provides idiot-proof removal. This scheme would permit any board to be safely removed at anytime. It requires a four level connector. The reset pin is moved to the fourth level guaranteeing that it is that last connection made and the first connection broken. The software implications of this are rather mind boggling, although if the error handling mechanism is robust only the tasks actively using the card would be affected.

Section 5

Software Interface

This section will eventually define exactly how ROMs are to implemented. It will include a complete list of defined resources, and the mechanism for their use. A large section will be dedicated to how the Wilson DMA system is to be migrated

At this point only the very basics have been defined. The firmware section contains two things. A brief description of how the directory structure differs from Nubus, and a description of the new routines defined to exist in this document.

There is also a section on error handling. This covers some system implication of special features provided by BLT.

Firmware

Format Block

The directory structure for BLT is lifted directly from Nubus with the exception of the top level format block. This has been modified to deal with the multitude of byte lane combinations possible with the BLT interface. The block is defined so that essentially any way that a ROM is connected will work.

When initializing a card, the CPU will read 16 bytes from the card, from the last sixteen byte locations of card space (ssss ssss ssss ssss). It will then examine these for valid byte lane combinations. In order to specify a byte address as valid, the card must assert a special value based upon the address. For the purpose of this description, the term lane defines the low three bits of the byte address used for accessing that byte. This corresponds to the byte lane if the card is 64 bits wide.

The special value is the low nibble asserted to byte lane and the high nibble asserted to it inverse. Thus a valid byte at the highest address would be specified as 87H(byte lane 7, 0111 in binary, its inverse, 1000 produces 1000 0111 of 87H). This pattern is used for the highest eight bytes. The lower eight bytes use the inverse of this pattern.

A card that asserted valid ROM data on all byte lanes would be read as the following

8	9	A	B	C	D	E	F
7	6	5	4	3	2	1	0

```
7 6 5 4 3 2 1 0
8 9 A B C D E F
```

This would consume 16 bytes of ROM.

This can be contrasted to a 64 bit interface that responded on only one byte lane

```
zz 9  zz zz zz zz zz zz
   6
zz 6  zz zz zz zz zz zz
   9
```

Where zz indicates a byte not driven by the card. This card is responding to byte lane 6. This encoding consumes 2 bytes of ROM.

As a final example, a 32 bit interface with ROM on a single byte lane, no tricks with the twist lines

```
zz zz A  zz zz zz E  zz
      5           1
zz zz 5  zz zz zz 1  zz
      A           E
```

This encoding requires four bytes of ROM. The card is responding to byte lane 1 of the 32 bit interface.

By use of this design any combination of byte lanes can be encoded. The CPU will use those byte lanes exclusively for ROM access. The rest of the format block is encoded as is shown in the figure 8.1.

Figure 5-1
Format Block Description (ESR)

As shown, the format is slightly different from that defined in Nubus. The features are all similar. The rest of the structure is defined exactly as for Nubus. Consult the Nubus cards and drivers

manual for a description. All offsets are defined in ROM bytes, which is independent of the number of byte lanes etcetera. The motherboard will convert to the proper address before accessing a resource.

Code within the ROM is defined to be 88K binary. No higher level language is defined. There does not seem to be a compelling reason to do anything more sophisticated.

Routines

This section will eventually contain a full description of the routines the system looks for in the card ROM. Obviously a large amount of this will be defined by the Pink folk. The current contents are the routines specified to exist within this document.

Figure 5-1
New BLT Support Routines

Enter Low Power	This routine does whatever actions are necessary to put the card into low power mode. This routine is called after the Low Power signal has been brought low, but before the CPU has actually "powered down"
Exit Low Power	This routine does whatever operations are necessary to bring the card back into normal operation. This routine is called only upon exit from low power mode, and is executed after the Low Power signal is brought high.
Card Disable	This routine realizes whatever operations the card requires for safe live removal. At the completion of this routine BLT will be disabled to the card and the reset line will be tristated, resetting the card.

The card that received a write error may also need some error recovery mechanism. This will likely be implemented in the interrupt routine, as that is how the motherboard is informed that it must do third party error servicing. All of these routines are optional.

BLT Support Routines

BLT requires some additional system support beyond just loading the ROM from cards. Software plays an integral role in initializing and configuring the interconnect.

Initialization

Software is responsible for bringing cards into operation. The obvious time when this occurs is at initial power up. The live insertion capability implies that software must also be able to bring a card up at any time. The first operation in either event is finding cards that exist.

The mechanism for finding card is very different from a conventional bus. BLT defines that a card asserts certain lines to certain states when in the reset state. Software has access to these values via the serial bus. A card is discovered by finding a defined pattern asserted in a node that is reset. In order to find cards that are inserted with the system running, software must occasionally (every couple of seconds) scan all nodes for new devices.

Once a new card has been identified it must be enabled. This is done through the serial bus as well. Software simply sets the card configuration bits in the appropriate registers and BLT is active for that node. At this point the card has a working bus, but is still in reset state. The last operation the CPU does is cleari the reset bit. Again this is done via the serial bus.

This should bring the card into operation. Some time should be allowed to pass (this will be specified) and then the ROM should be found. This is done by reading the upper most 128 bits of the cards address space. Just as in Nubus, the patterns detected define how the ROM is connected. This allows software to read the ROM without defining how a card must implement it.

From this point on the process is identical to Nubus. The test pattern is checked to verify that the card really is there and working properly. If everything is okay, drivers can be loaded and the cards firmware initialization completed.

If no valid information is found, then the card ROM is probably incorrect or nonexistent. The node should NOT be disabled. The system will refuse to admit it is there, but tasks must be able to access it. Hardware designers would shoot us if they had to have a working ROM to have any access to their card.

The live insertion brings up an interesting point. Can an application be informed of a new device being inserted or can this be done only at application start up?

Card Disable

The card disable software provides a clean method of removing a card. The operations necessary for the card itself are fairly simple. First the card disable routine must be called. This will allow the card to take any required actions such as saving state. Next the card is reset by assertion of the reset bit. Finally the BLT interface is disabled by clearing the configuration bits.

The more challenging aspect of card disable is dealing with applications. Each must have some mechanism by which a resource they are using can go away. This could range from the application dying to something elegant like executing a software fallback for a renderer. This same ability is necessary for error handling.

The functionality of card disable should be provided to the user through something equivalent of a control panel that is always accessible to the user.

This is actually quite similar to the shutdown item on the Macintosh. It allows all applications to take whatever action is necessary to prepare for this catastrophic event.

Error Handling

The error handling model in Jaguar must differ from Macintosh in one important way, elegance. At the very least, an error in hardware should only impact the tasks using that piece of

hardware. In no way should the loss of a card crash the system entirely. This is particularly important given the integration of the phone with the machine. The phone should be available all the time.

The error handling model on BLT is very limited in order to simplify the system as a whole. When a card dies, that particular node is the only one effected. All subsequent packets are errored internally to BLT. System software must take whatever actions are appropriate for hardware. The standard model would be to reset the card the first time it fails, and disable it the second. Better, the user could be asked after some ridiculous number of failures. The bottom line is that the error model defines that an error is ultimately fatal to the card.

As discussed in the card disable section, this brings up an interesting problem for the system; what to do with tasks that were using the failed device? The best option seems to be to execute a method that allows the application to either terminate (the default) or take some more elegant action.

Wilson

Wilson is discussed in detail in its own section of this ERS. In its current form it is an extremely powerful piece of hardware, very useful on the motherboard alone. More important is that it is a destributed DMA architecture based on block writes.

There are several basic ideas that went into the design of Wilson. It was to enable a separation of sources and destinations into orthogonal functional blocks. In English, this means that the complexity a card introduces into a system is dealt with by that card itself. For example the motherboard has virtual memory implemented via pages. This can cause a single block of data to be scattered throughout memory. The motherboard must implement the conversion mechanism from physical address space to this virtual space. This allows cards to use very simple physical addressing to a contiguous space, no matter how the data is stored. Another example would be a card that implements a color look-up table instead of direct color. All pixel transfers on Jaguar are defined in terms of direct color. The destination card must take whatever action is necessary to convert this format to its internal format.

There is another aspect to the separation of source and destination. Every card added to a system adds some set of features. A video input card allows video to be entered into the system. A frame buffer allows graphic display. As much as possible, the capabilities a card has should be defined by the card function, not by what has to be done to make it work. Video input and graphic output have no fundamental connection, yet almost every card on the market that does real-time video in has a built in frame buffer. This is done simply because the bandwidth of video is too high for most busses. BLT solves this problem but that is only half the battle. In order to truly separate the two halves, video input and graphic output, a protocol must be defined up front that both can conform to. This is exactly what Wilson does. It defines the fundamental protocols that are necessary for device independent graphic transfers. All devices that produce pixels (renderers, video capture, decompression, etc.) convert the pixels to a defined format. All destinations (frame buffers, memory, compression, etc.) accept that defined format.

Another idea that Wilson is based on is that write operations are much higher performance than reads. As mentioned before, BLT is much better at writing than at reading. This is actually true of almost all transfer mechanisms, from busses to networks to whatever. Wilson is based entirely on a write model. This causes Wilson to be a distributed DMA system. A card writes its data across BLT

to a destination[1]. This write occurs in a fixed format. Each card implements a small piece of the Wilson system. This makes its capabilities grow along with the system.

A further idea of the Wilson system was coherent expansion. A card can be added to the system that adds a new "resource" to Wilson. Video in, compression, resizing, rendering, and compositing are some graphic functions that could be added. Each of these are peers to the functionality provided on the motherboard. They are all accessed by the same model, and all share common formats. Wilson is not only a DMA system, but an expandable processing engine.

Further, Wilson is in no way limited to graphics. The functionality implemented on the motherboard is almost exclusively graphics oriented, but this is simply because that is the functionality that was needed. The Wilson system defines the interconnect mechanism, it makes no comment on the particular data transfered. This is done by the resources being used. BLT and Wilson could form the foundation of a world class frequency analyzer, a powerful image processor, or any other data processor.

All of these capabilities are realized with one simple mechanism, streams. The Wilson section discusses its use of streams. This section previously covered how BLT implements them. They exist so that extension devices can cleanly plug into the Wilson system and increase the performance of the entire machine.

The hardware to implement streams from cards is defined in this document. What is as yet undefined is the mechanism for extending the Wilson software architecture to include extension devices. This will be a long and involved development. Eventually, the Jaguar cards and drivers manual will have a large section on how to build a card and write a ROM that is Wilson compatible. At that point a user can plug a third party frame grabber in, and get a clipped, alpha blended, video window on the motherboard display.

---

[1]It should be noted that the Wilson chip on the motherboard is capable of doing read operations because realism says some cards will not implement Wilson. This mechanism is much lower performance that writes.

Section 6

BLT Hardware Implementation

A fairly comprehensive implementation specification exists for BLT. It is currently about a month out of date so it does not reflect some recent changes to error handling, streams, and pinouts. It will be included here when it has been revised. The current version is available for anyone interested.

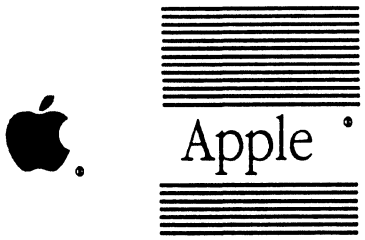
Section 7

Issues

The error handling mechanism is an area of great concern. Is error recovery necessary or useful?

The model for promotion is that the increased priority of a packet just appears without out any indication. Is this acceptable? Should promotion even be presented to a card?

The pattern in which skip presents packets is a mess, particularly in regard to priority packets. What stream is presented after a priority packet is skipped? Is it always stream 0 (the lowest available) or is it the stream subsequent to the last stream dealt with? Does it matter?



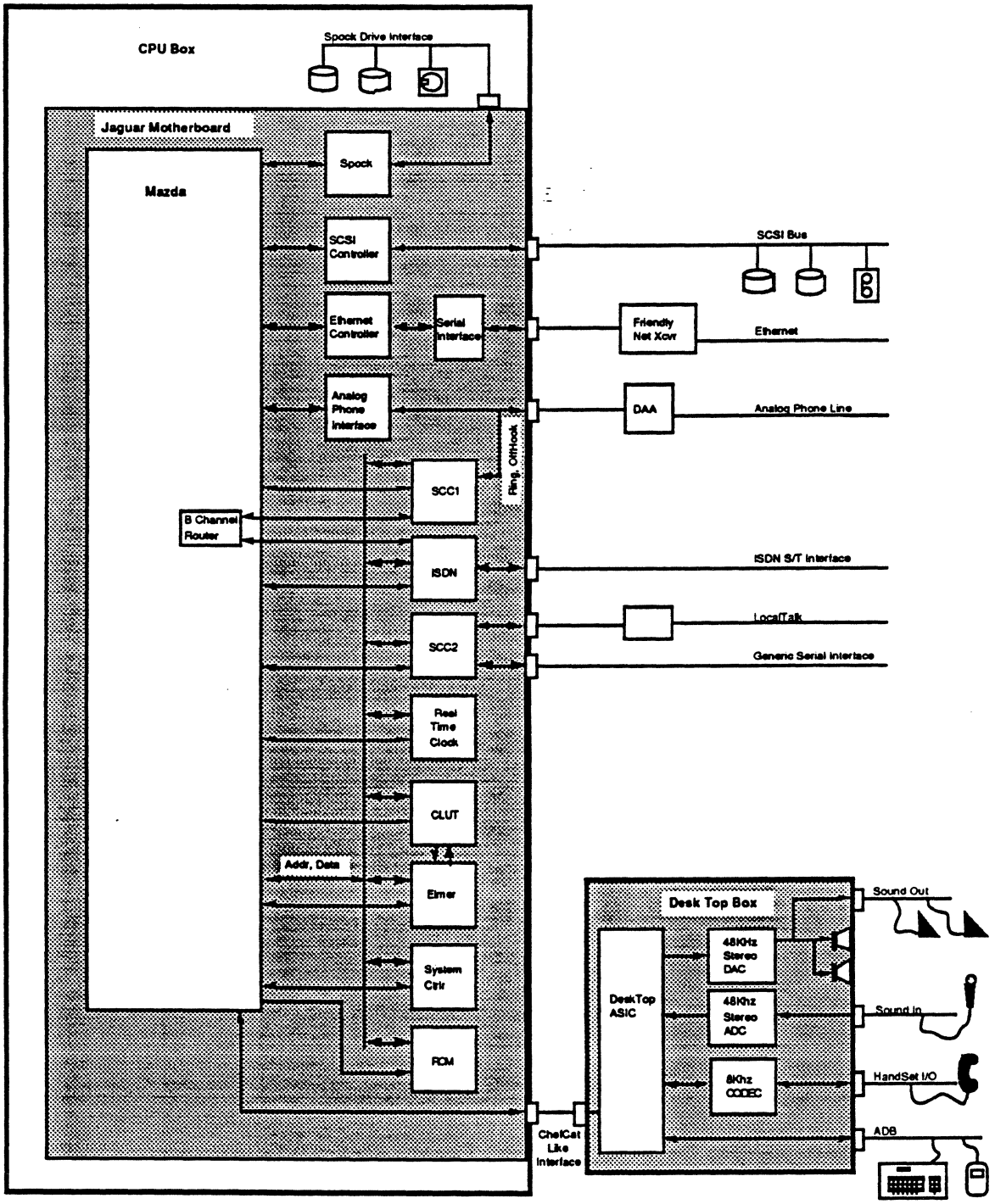
Jaguar I/O Subsystem

External Reference Specification
Special Projects

Nov. 13, 1989

Return Comments to:
the I/O boys: Bill, Bruce, Chris, Jim , Jeff, Kevin
Phone: Bill x9047, Bruce x6502, Chris x0671, Jim x0672 , Jeff x5832, Kevin x0308
AppleLink: NICHOLS1
MS: 60-AA

Contents



Jaguar I/O Subsystem

Jaguar I/O Subsystem Diagram

Section 1

Introduction

This chapter deals with the I/O Subsystem of Jaguar. Before describing the details of the proposed implementation, it is perhaps useful to discuss the ideas which lead to the current design.

This chapter deals with the I/O Subsystem of Jaguar. Before describing the details of the proposed implementation, it is perhaps useful to discuss the ideas which lead to the current design.

What is I/O?

Before one addresses how to handle I/O, one should have a clear idea of what I/O is considered to be. The reason for asking this seemingly trivial question revolves around how to treat BLT within the Jaguar architecture.

What probably comes to mind immediately when one mentions I/O is the process of moving data between the "outside" world and the processing component of a system. Some obvious examples are reading/writing a hard disc and transporting packets over EtherNet.

However, the picture is somewhat less clear when one views something like NuBus. If someone asks me whether I consider NuBus part of the I/O system, the answer is probably: it depends! When it is being used to support a dumb I/O device, NuBus is part of the I/O system, being nothing more than a way of accessing the I/O support chips. However, if a NuBus card contains intelligence, NuBus could be considered as a simple extension of the processor/memory bus, and thus, not part of the I/O system. When one looks at the system from a higher level, however, that intelligent card may be a network card. Thus, when viewed from a sufficiently high level, that NuBus card is providing I/O services.

If one takes to an extremely low level view of a system, there may be no I/O directly visible. For example, if one looks at a Mac from the CPU bus level, there is no I/O! There are just some chips which exist in the address space that have some funny side-effects. If software accesses these chips in exactly the right way, some "I/O" may happen. The point of this obfuscation is to raise the reader's consciousness to the fact that the concept of "I/O" depends to a large extent on one's perspective. It also points out that on the Mac of today, most of what the user considers I/O is being managed mostly by software, not hardware.

With the above discussion in mind, Jaguar has divided the expansion world into two different pieces. The I/O Subsystem (i.e., Mazda and its related I/O chips) deals with discs, built-in networks, etc. BLT deals with expansion cards, some of which may be performing I/O services; however, these cards are considered as a different class, separate from the built-in I/O devices.

What is NOT I/O?

Within Jag1, the Video Decompression and BLT slots are not in the same category as, for example, a SCSI disc. The I/O Subsystem (and,hence, this chapter) deals only with the interfaces which are directly managed by Mazda.

Atomic Unit of Transfer

Upon examining any of the I/O devices, one can quickly identify a unit of transfer below which it makes no sense to discuss. For example, in the SCSI hard disc world, a single transaction consists of the command sequence (6-10 bytes) followed by a data transfer stream which is at least a sector. It is senseless to discuss individual bytes of the transaction. Another example is the LocalTalk packet, whose transmission actually consists of 3 separate HDLC frames over the wire, with an associated protocol. For the purpose of discussion, we will call this lowest level unit the Atomic Unit of Transfer (AUT).

Note that for some devices (e.g., the keyboard), the atomic unit of transfer may be a single byte. There are also some devices for which it makes sense to talk about individual bytes, but which are often treated in groups anyway. For example, even though the ASYNC serial interface deals with single bytes, they may be logically grouped when, for example, an XMODEM transfer is taking place. And, there may be devices for which an AUT is 1 byte, but which, for efficiency reasons, may be grouped if several bytes come in within a small time window; e.g., the case of ASYNC output, where the source of output may be a line of data which is presented to the I/O system as a unit.

The point of introducing this AUT concept is that to the relatively low-level I/O driver, it is the AUT which would ideally by handled by the hardware sub-system. I.e., the SCSI driver would like to present a transaction request to the hardware, allow the processor to be used for other work and be informed later, with a single interrupt, when the entire transaction is complete. The AUT is the smallest unit with which software should (ideally) be expected to deal.

Devices of Interest

Before discussing how to manage the I/O devices, we should first list the devices which are potentially of interest in the Jaguar world. Here are some of the devices of interest, along with an idea of their AUT sizes:

KeyBoard/Mouse: The keyboard and mouse have very small AUTs; as supported on ADB today, each transfers 2 bytes per transaction.

Hard Disc: The smallest AUT is typically a disc sector; 512-1024 bytes.

MO Drive: Like the Hard Disc, the AUT is a sector.

Floppies: Again, the AUT would be a sector.

CD Quality Stereo Input: The unit here depends upon the response time required. The absolute smallest AUT would be 16 bits per channel, or 4 bytes.

CD Quality Stereo Output: The unit here depends upon the buffering capabilities of the DACs; assume the same size as for input, 4 bytes.

Serial (RS-232/432): 1 byte AUT; however, many protocols (e.g., XMODEM, Kermit) would use a higher-level AUT which would be on order of 128 bytes.

LocalTalk: The AUT here is an entire packet; this is a maximum of 600 bytes, more typically 512 + protocol.

LAN: The AUT is a packet; on the order of 500-1000 bytes.

ISDN: Since the data transfer protocol is HDLC, the AUT would like to be the HDLC packet; this is somewhat dependent upon such things as the lower level protocols of the chip set to be used.

Analog Phone: Except for small sequences to exercise the call establishment, etc. most of the real data would look like sound output.

Jaguar I/O Subsystem

Since Apple is moving in the direction of multi-tasking, we are assuming that the ideal I/O system would not require any CPU processing for the AUT. To achieve this with the lowest cost, Jag1 handles I/O by means of a single chip controller (Mazda) which combines a programmable I/O Engine (Wankel) with a multi-channel DMA mechanism.

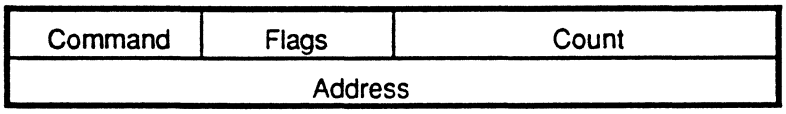
However, the basic programming model for Jaguar I/O can be discussed in a somewhat abstract manner. In a lower priced Jaguar model, some of the I/O processing may be managed more explicitly than on Jag1. Because of this, the maximum number of (really) simultaneous transfers may be reduced. We believe that the model presented below gives sufficient flexibility for future implementations.

Architectural Overview

The basic I/O model proposed for Jaguar is that of a multi-channel "DMA". The interface between the software and the "hardware" is by means of Channel Programs which are created by the software (e.g., Pink Access Managers) and "executed" by the hardware (e.g., Mazda). Results of channel commands are returned to the software at interrupt time to indicate any errors, short records, etc. which may have occurred. After initiating a Channel Program, software intervention is not required until the command sequence is finished.

Note that the term "hardware" above refers to the combination of hardware and software of an implementation. In particular, a high-end system -- like our first box -- would have all of a Channel Program's execution performed in hardware (within the I/O Controller chip). However, a low-end system may make tradeoffs, even to the extreme of having all of the channel command processing being done by software.

The Channel Programs are sequences of Channel Command , which are (aligned) 8-byte "instructions" to the "channel processor". Each command word consists of:



The Command byte is interpreted by the channel processor (Wankel); it would typically be something like "Read a Block", "Write Commands to Device", etc. The Flags byte is used to specify sequencing options (e.g, DataChain). The Count and Address fields define where the associated data for the command is located.

Note that the address is a physical address; the system software must perform the necessary page locking, page mapping, etc. before initiating a Channel Program.

The Ready and/or DataChain flag bits allow sequences of Channel Commands to be formed into a channel "program". Many Channel Programs simply consist of a vector of channel command words. In order to provide more flexibility in the organization of Channel Programs, a Branch command is provided whose Address field specifies the new start of a Channel Command sequence; this provides a "branching" capability for Channel Programs.

This "branching" capability is provided as a convenience to link separate vectors of Channel Programs into one logical sequence. Unlike normal programming, there are no explicit conditions which can be tested, so there is no equivalent of a conditional branch in the architecture.

As a simple example of how Channel Programs are used, take the case of a SCSI transfer. The application will make a call to the system asking for data (say, from a file). The file system , in turn, figures out where in the file (and which device) to relevant data exists and calls the disc's Access Manager. The Access Manager locks the relevant pages, obtains the corresponding physical addresses and creates a Channel Program. The driver then informs the I/O Controller of the desire to initiate the new Channel Program. After initiating the operation, the XJS is completely free to continue processing, with no overhead until the controller signals that the Channel Program is complete.

Asynchronous Input

One class of I/O does not, unfortunately, fall within this simple model; namely, unsolicited and/or asynchronous input. Examples of this are network input, keyboard and mouse events, etc. The reason that these cause problems is that they occur at unexpected times and an arbitrary number may occur before the CPU can get around to processing them. Unlike, for example, reads from a disc, input data from the keyboard is not explicitly requested.

In terms of the channel program paradigm, these inputs have two sticky issues:

a) How does the controller know where to put the data?
b) How does it inform the CPU of where and how much data arrived?

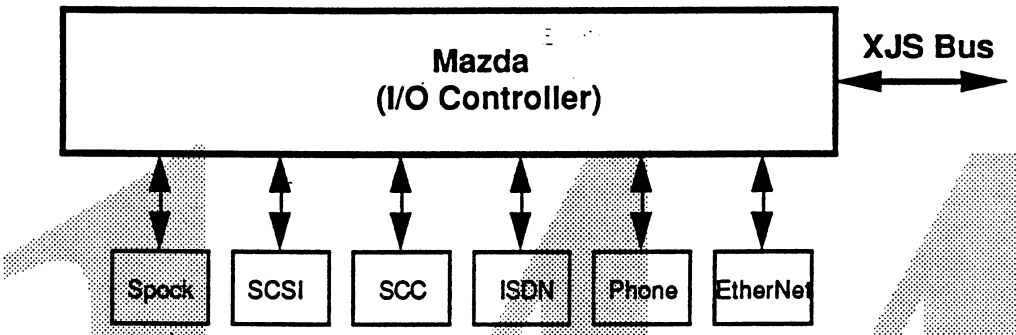
The solution to this class of transfer is to use Channel Programs to establish buffer areas using normal Read commands. By setting up a sequence of such Reads, an arbitrary number of buffers can be established to handle the worst case latency requirements (i.e., the maximum number of input events which could be expected to arrive before the first ones are handled and the buffers reused). In addition, the Flag byte of the each Channel Command can contain an "update" bit which will cause the final value of the Address and/or Count fields to be updated to the original Channel Command (thus, overwriting it) which will indicate the length of the incoming data.

Jag1 Implementation

On the Jag1,the I/O Controller chip implements all of the Channel Program processing. Instead of having this all done by "hardware" in the sense of state machines, etc., we are proposing that the chip include a processor which implements all of the command interpretation, results posting, etc. under "program" control. Hardware resources are applied where the actions are well defined and, hence, easier to guarantee "correctness".

Mazda, the Chip

All of the I/O within Jag1 is controlled by a single controller chip (Mazda) whose over-simplified place in the system can be represented as in the following diagram.

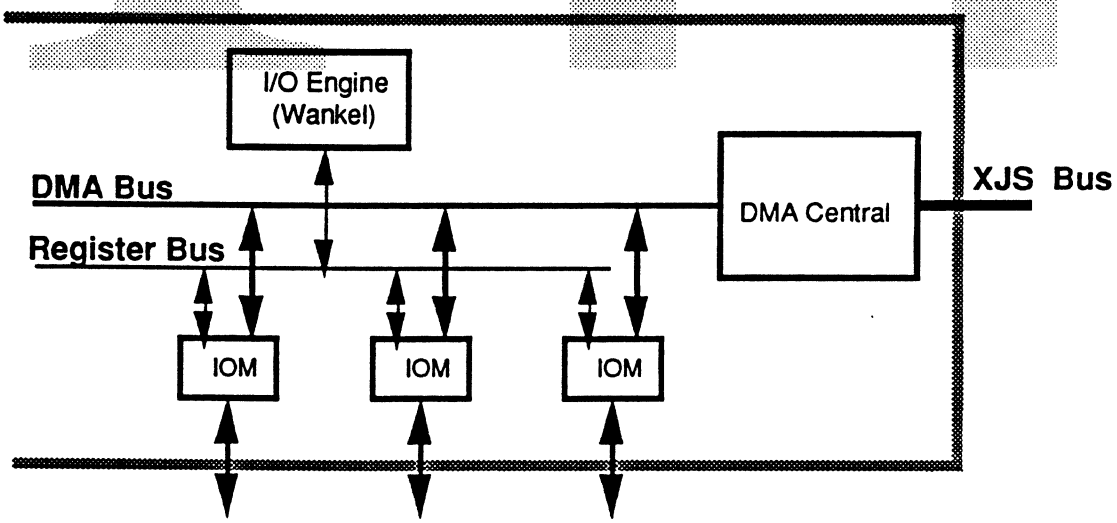


The smaller chips implied in the above diagram are the low-level blocks which are managed by Mazda. While some of them correspond to single chips (e.g., SCC or SCSI), some blocks (e.g., ISDN) are actual composed of multiple chips, including some support within Mazda.

One important point in this architecture is that XJS programs do not have direct access to the devices managed by Mazda. It may be possible to provide some channel protocol to allow some devices (e.g., SCC?) to be accessed a somewhat lower level (e.g., accessing SCC registers). However, we would like to maintain the highest level of abstraction and add such "hacks" only if absolutely necessary.

Inside Mazda

In a highly abstract model, the chip's organization looks like the following:



Note that while this diagram shows how the parts logically interact, it does not necessary correspond to the actual partitioning within the chip.

Mazda is composed of three types of blocks. Wankel is the processor section; it is composed of an 8-bit data, 16-bit instruction CPU. Details on this engine are described later. The DMA Central block comprises a shared XJS bus interface which manages all interactions with the System Interconnect. The third type of element are the I/O Modules (IOM), which contain the individual hardware logic for each specific device interface.

Wankel, the I/O Engine

Wankel is probably the most interesting part of the chip. It consists of a simple processor with one unique design criteria; it implements task context switching in hardware (one clock!). Thus, the code for the engine consists of a set of tasks (at least one for each I/O device) which is designed to yield to other tasks whenever the task needs to wait for something (e.g., for a read from the I/O device to complete). In general, under "idle" or "waiting" conditions, each task requires 3 clocks. Assuming that the engine is running at 1/2 of the processor clock (thus yielding a 25 mip engine!), this task overhead is only 150 (120) nsec. Note: the code must be written to sprinkle "swtask" instructions in order to guarantee latency requirements of the worst case device.

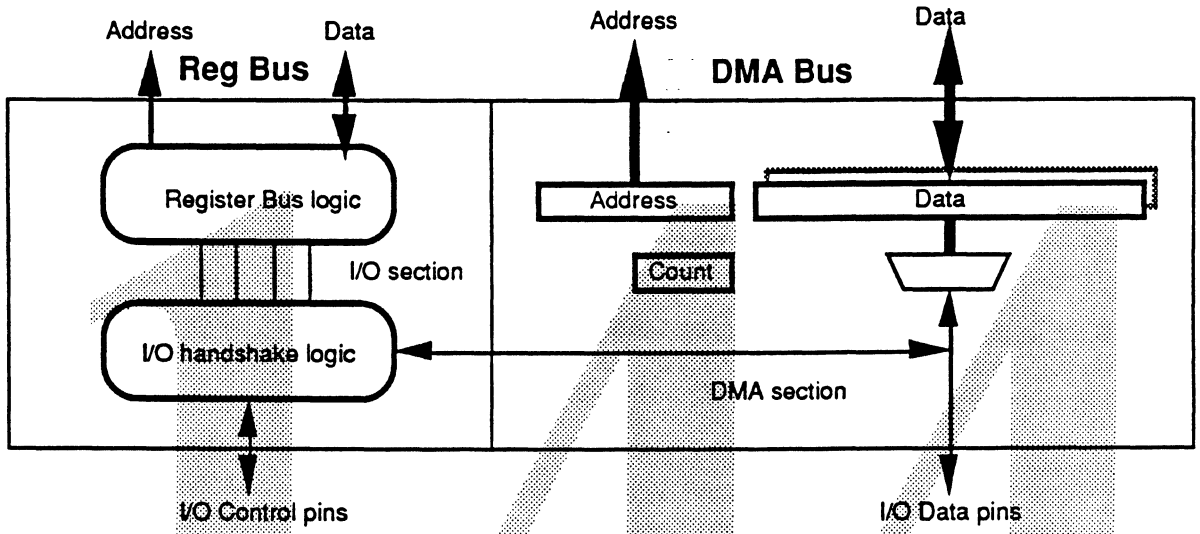
The point of providing such an engine is that every device is being serviced "simultaneously". Thus, instead of relying on the (serial) services of the XJS to drive the devices, with latencies of each device adding up, all the devices are running at full speed at the same time, freeing up the XJS to do useful work.

In addition to managing the Channel Command processing, the Wankel also implements the actual "commands" for those devices which require non-trivial software interactions. The messiest of these devices is probably the SCC. The abstraction of the Channel Programs allows the system side of the world to think in terms of LocalTalk packets. Wankel implements this abstraction by performing all of the Link Access Protocol (which is incredibly messy!) itself.

Even though Wankel is very fast, it would not be able to keep up with the worst case transfers of data from all devices simultaneously. For normal data transfers (i.e., where no special processing is required), Mazda's DMA channels perform the actual transfer of data between the device and memory. While the DMA transfers are going on, Wankel's sole job is to manage the chaining of Channel Programs and watching for termination conditions from the device. Thus, the transfer rate of a device is dictated by the it and the memory system, not the speed of Wankel.

I/O Modules (IOMs)

The I/O Modules all have the following general organization:



The DMA and Register Bus Logic sections would be almost identical for each IOM. The "uniqueness" of an IOM is (primarily) contained within the I/O Handshake Logic portion. The idea is that each I/O device with which an IOM interfaces has different handshake characteristics. For example, the SCC has the infamous 6 PCLK + 200 nsec. rule about back to back accesses. These sorts of device dependent anomalies are managed (in hardware) within the handshake logic. The advantage of this model is that the Wankel coding for each device does not have to worry about these idiosyncrasies, which makes the writing and debugging of the interface code much simpler and less error prone.

Mazda is designed under the assumption that most transfers contain a major part which can be managed very simply by straightforward DMA logic (via the DMA Section). However, for all of the special processing (e.g., SCSI bus arbitration, disconnect, etc. or LocalTalk Link Access Protocol), the Wankel gets involved. Note that the bandwidth requirements of the I/O devices are not limited by the processing power of the I/O Engine; the DMA logic provides maximum bandwidth without any direct involvement of the I/O Engine (after the DMA operation is started).

Device interrupts are managed by Wankel. In most of the devices, the occurrence of an I/O interrupt does not signify that a whole transfer is complete, merely that the next stage must be executed. For example, today's SCSI manager (and/or device driver) receives multiple interrupts (e.g.,when a device re-connects). In the Jaguar architecture, a CPU is interrupted only upon completion of a Channel Program (unless a Channel Command requests an immediate update, such as for asynchronous input events).

Miscellaneous Mazda Support

In addition to providing support for "classical" I/O devices, Mazda also contains logic for the following:

Sample Rate Conversion

To support the processing of (primarily, audio) data streams which are sampled at one rate, but whose processing assumes a different sampling rate, some sort of Sample Rate Conversion must be performed. While the XJS could certainly perform these calculations, separate hardware is being proposed in order to off-load the XJS(s) from this trivial, but tedious task. Since Mazda already has the logic to fetch and/or store streams of data (DMA), it was deemed reasonable to include such support within Mazda, even though it is not strictly "I/O".

Booting Support

Mazda is the only chip which has direct access to the "System ROM" (or, EEROM). During initial power-up (or, reset), Mazda controls the first several stages of the "booting" process. (This is covered in more detail in the System Environment chapter.)

Mazda controls the RESET lines which go to (at least) the XJSs. Upon power-up (or, the RESET button), the XJSs remain in reset state until Mazda is able to copy a boot image into Main Memory. After this, Mazda will remove RESET to the XJSs, allowing them to continue the boot process.

Reduced Power Control

Mazda will perform whatever processing necessary to cause the system to enter a reduced power mode, where the system can quickly respond to, for example, the phone, while consuming much less power than under normal operation.

Interrupts

Mazda contains the centralized interrupt logic for Jag1. In addition to internally generated interrupts (e.g., Channel Program completion), several other hardware interfaces can request interrupt services (e.g.,cards in BLT slots). The hardware in Mazda is designed to present a single interface for Interrupt Service Routines (ISRs).

Issues

There are numerous issues open to discussion. Please refer to the "Issues" section at the end of this document for a compilation of issues by subsystem.

Section 2

I/O Architecture

Terminology

In order to improve the clarity of the following discussion, the following terms are defined:

Access Manager: an XJS process which is responsible for managing access to a particular device or interface. This process is roughly analagous to what is known as a device driver in conventional systems.

Interrupt Service Routine (ISR): a section of code which is installed in the kernel, and is invoked when an interrupt from a particular device is detected. (Ideally, the ISR's would do very little processing before sending a message to an Access Manager, and exiting. It is a goal of the Jaguar I/O subsystem to ensure that this is practical.)

Channel Program: a data structure created by an Access Manager in XJS memory space which describes a series of I/O operations to be performed.

Channel Command : a 64 bit element of a Channel Program which describes an individual I/O operation. e.g. "write 512 bytes of data from real memory address 0x00043000 to the device".

Channel Program Pointer (CPP): a resource internal to the Mazda I/O processor which contains the address of the currently executing Channel Command within a particular Channel Program. This pointer is written by an Access Manager when initiating a Channel Program, and is updated by Mazda as it sequences through the Channel Commands within the channel program.

Wankel Task: a section of code which is executed by Mazda's "Wankel" processor. Wankel tasks manage the physical interfaces, and interpret the channel programs which use those interfaces. For example, a single Wankel Task manages the SCSI interface, and interprets multiple channel programs which control the individual devices connected to the SCSI bus.

Channel Programs

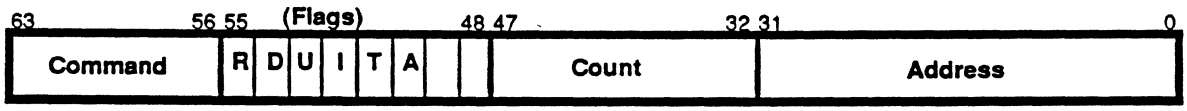
Channel Program Organization

An important goal of the I/O subsystem is to minimize the impact of interrupt and other latencies on system performance. Mazda's architecture aids in achieving this goal by providing facilities which make it straightforward to organize channel programs as queues of tasks to be performed by Mazda. By organizing the channel programs as queues, performance is improved by overlapping Mazda's processing of a queued task with the interrupt and processing time which is incurred on XJS when an I/O operation completes.

In situations where this queued structure provides little benefit, simple linear channel programs can also be used.

Channel Command Definition

Shown below is the format of a Channel Command:



The fields are defined as follows:

Command This field is interpreted by the Wankel task. It would typically have a meaning such as "Write Commands to Device". Since each Wankel task sees Channel Commands only for a specific I/O channel, the decode of the Command field may be unique for each task. Therefore, the same value in the Command field may represent a different command to each task.

In addition to device specific commands, a command to jump (unconditionally) within the channel program will be implemented by all Wankel Tasks. In such a Channel Command the address field will contain the address of the next channel word to be executed. This jump command allows a loop to be created within a Channel Program, so that it can be organized as a queue (circular buffer) of tasks to be performed by the device. (See channel program examples.)

Flags The Flags field is also interpreted by the Wankel task. Typically this field will be interpreted by the Wankel task after the Channel Command has been executed to determine what action to take next. The following flags have been defined:

Ready This bit is used to pass ownership of a Channel Command, and the buffer it points to between XJS, and Mazda. It is set by XJS, and cleared by Mazda. A zero in this bit indicates ownership by XJS (i.e. not "Ready" for processing by Mazda), a one indicates ownership by

Mazda. For example, during a Read operation, after an XJS Access Manager had set up the Channel Command to point to an available input buffer, the bit would be set to one, indicating that the buffer was available for Mazda to write received data into. After Mazda had filled the buffer, it would write the updated Channel Command back to memory, with the Ready bit reset, giving ownership of the Channel Command and buffer back to XJS.

Mazda will suspend channel program execution when it encounters a Channel Command with the ready bit cleared. (In order to activate a new channel program, the first Channel Command must have the Ready bit set).

Both Mazda and XJS must refrain from writing to a buffer or Channel Command until the other processor has relinquished ownership.

DataChain If set, this flag indicates that the current Channel Command is actually not complete, but must be continued across another block of data. Data chaining is used whenever the data for a transfer does not all reside contiguously in memory or the amount of data exceeds the size of the Count field (64 KB). The Chain Data flag signals the Wankel task to fetch the next Channel Command but to treat it as a continuation of the ongoing data transfer.

In the case of unsolicited reads (e.g. Ethernet reception) a sequence of Channel Commands could be created each pointing to small buffer segments. Each of these Channel Commands would have the DataChain bit set. When an incoming packet spanned several of these buffers, the DataChain bit would be cleared when the Channel Command corresponding to the last buffer segment was written back to memory, indicating to XJS that the End Of Packet had been reached.

Update This bit indicates that the updated Channel Command should be written back to memory after the operation completes. Typically, an access manager will set the Update bit in all Channel Commands so that the Ready bit can be reset by Mazda. However, if the access manager does not want the channel program to be modified, it would would not set the ready bit. This might be the case with a simple channel program which generates a single interrupt, and which the access manager might want to execute again simply by rewriting its address into the CPP register.

Interrupt Setting this flag indicates that XJS is to be interrupted upon completion of the Channel Command. If a sequence of Channel Command are needed to perform an operation (e.g. Write Control,

Write Data, Read Status) the Access Manager might want to set this bit only in the last Channel Command of the sequence.

TimerInt This bit indicates a special mode of operation is to be used for the transfer. In this mode, the updated Channel Command is written back to memory and an interrupt is generated, if any data transfer has taken place between ticks of an internal timer. However, the Channel Command remains active until the Count is exhausted, or the Channel Command is aborted. This mechanism allows the Access Manager for an asynchronous serial port to receive interrupts at intervals greater than every character, while guaranteeing timely response to input characters. [This bit will probably be removed from the flags field as this is not a terribly general operation. A Wankel task can perform this operation in implementing a particular command code. Ed.]

Aborted When Mazda responds to a request to abort a channel program, it will acknowledge the abort by writing the active word of the channel program back to memory with this bit set, and interrupting XJS. This bit will also be set by Mazda if the controlling Wankel task decides to abort the channel program due to a serious error. (See the section on Aborting Channel Programs.)

Count This field of the Channel Command specifies the the length of the associated data buffer. As the transaction proceeds, an internal representation of this count is decremented until it goes to zero, or the transaction completes by other means. If the Update bit is set in the Flags field, the residual count will be written back to system memory, so the Access Manager can determine how many bytes were actually read or written to the device.

Address This field specifies where the associated data buffer is located in XJS memory. Note that this is a physical address, and the Access Manager must perform the necessary page locking, and virtual to real address translation before initiating the operation.

It is also possible for channel commands to utilize an immediate form of adressing, by using this field to supply data for reads or writes. If the field is to be used to return read data, the Update bit will have to be set in the Channel Command. Since the Wankel processor has the ability to read and write the Channel Command words, this feature is implemented simply by the actions taken by the Wankel Task in interpreting a command code.

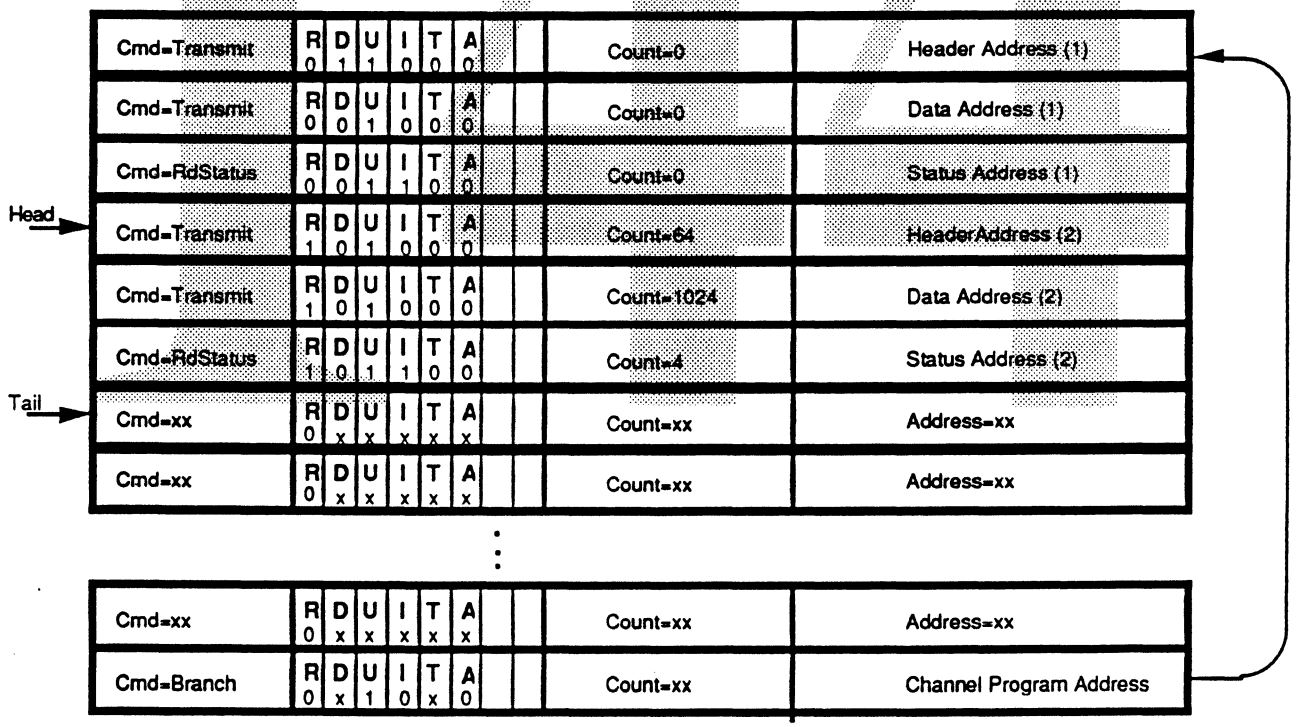
Channel Program Pointers

The channel program pointers are registers within Mazda which XJS can write to via memory-mapped I/O write operations. Whenever an Access Manager adds a Channel Command to a channel program it should write the address of that entry to the CPP register associated with that channel program. If the channel is currently idle it will resume execution of the channel program at the specified address. However, if the channel is executing a Channel Command, the current CPP register will not be overwritten - the new Channel Command will be executed once Mazda sequences to that word.

[See "Mazda Memory Map" in the Hardware Implementation section for the location of the CPP registers.]

Channel Program Examples

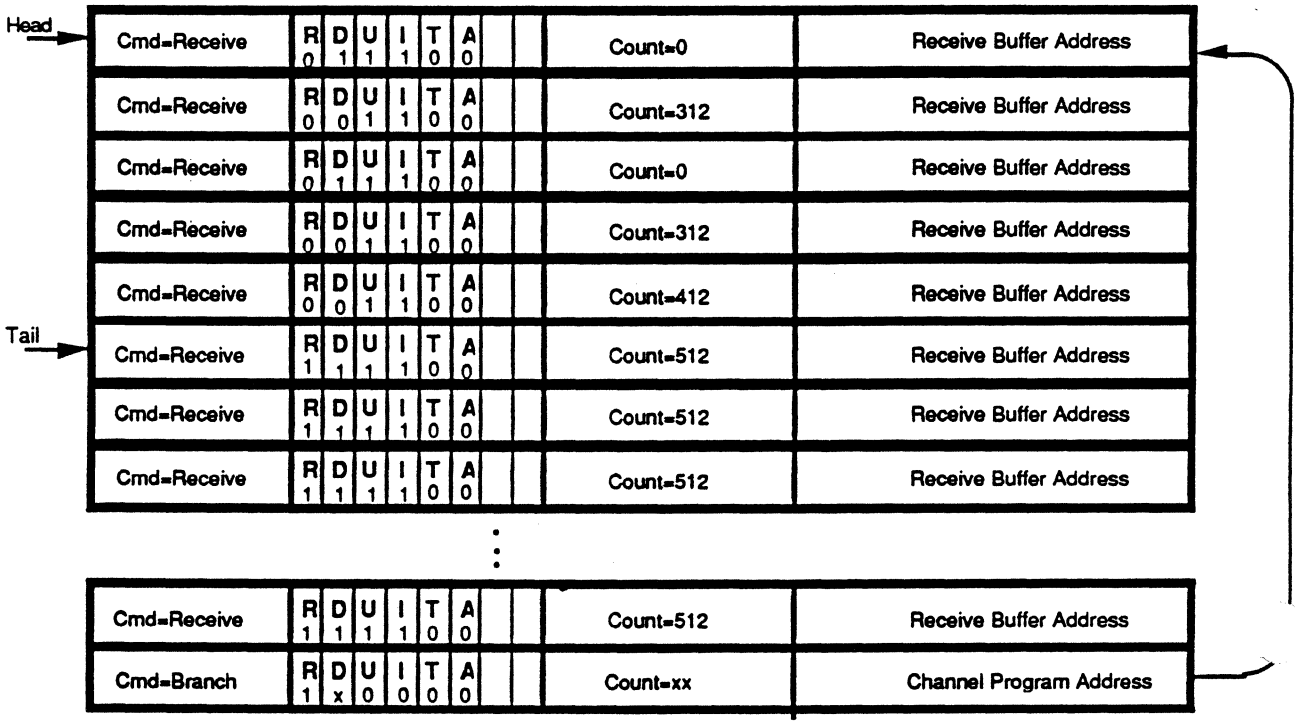
Ethernet Transmit



Shown above is an example of what an Ethernet transmit channel program might look like. The channel program is organized as a queue. The access manager has written 6 channel words, corresponding to 2 transmit frames, and is ready to write the next channel word at the tail of the queue in the 7th entry. At the time of this snapshot, Mazda had completed execution of the 3 Channel Commands corresponding to the first transmit frame, and is processing the Channel Command at the current head of the queue in the 4th location. Notice that Mazda has reset the

Ready bits in the Channel Commands it has completed, and has decremented the counts to zero, indicating that the requested number of bytes have been transferred.

Ethernet Receive



Shown above is an example of what an Ethernet Receive channel program might look like. Again, this is a snapshot of the channel program partway through the processing. Initially, the access manager had set up each word in the channel program to indicate a read into a 512 byte buffer segment, and had set the DataChain, and Ready flags in each of them. This figure indicates indicates the state of the channel program after Mazda receives two 712 byte frames, and one 100 byte frame. Mazda is ready to receive the next packet into the buffer pointed to by the Channel Command at the tail of the queue. XJS has not yet processed and relinquished the Channel Commands at the head of the queue.

The 712 byte frames each spanned two buffers. The first buffer has a residual count of zero, and has the Data chain bit set. The second buffer has a residual count of 312, and has the DataChain bit cleared, indicating that the last 200 bytes of a frame were stored in the corresponding data buffer. The short 100 byte frame required only a single Channel Command and data buffer. In this Channel Command the DataChain bit has been cleared by Mazda, and the residual count of 412 indicates that a 100 byte packet was received.

In order for an access manager to get status information concerning a received Ethernet packet, it may be necessary for the Wankel task which is managing the Ethernet interface to append a fixed number of status bytes to the data. The access manager could then strip these off, before passing the data on to its client process.

While it is likely that the access manager would only wish to receive an interrupt after an entire packet had been received, there is no way of knowing ahead of time which Channel Command would be associated with the last segment of a frame. Therefore, the Wankel task must ignore the Interrupt bit in the Channel Command, and either interrupt on each Channel Command completion or on packet completions.

Aborting Channel Programs

There will be circumstances in which an Access Manager will want to abort an executing channel program, and halt activity of the channel (for example if a timeout occurs). A mechanism to do this will be provided through a special channel program used to communicate with Mazda's internal Wankel processor. A special "Mazda Access Manager" will be required to manage the channel program used to send these abort messages.

The mechanism for performing the abort would be as follows: The Access Manager for a device would send a message to the Mazda Access Manager, requesting an abort of a specific channel program. The Mazda Access manager would add a Channel Command requesting the abort to the special Wankel channel program. Upon receiving the abort command, Wankel would force the appropriate I/O module to abort its current Channel Command. At this point the Channel Command would be written back to memory with the Aborted bit set, and an interrupt would be generated for the channel in question. The abort bit provides acknowledgement to the device Access Manager that the abort was performed.

Additionally, a channel program may abort itself if a serious error condition is detected by the I/O module or by the Wankel Task which is controlling it. In this case, the Aborted bit would also be set in the current Channel Command before it is written back to memory, and an interrupt is sent to XJS. After terminating the current command in this way, the Wankel task would go into its idle state, and stop executing any further Channel Commands.

[Format of abort Channel Command to be included here.]

Asynchronous Events

Mazda will provide an additional channel program resource to allow I/O modules to report low frequency asychronous events. This shared channel program alleviates the need for an additional channel program to be allocated to each I/O module which needs to report asynchronous events. However, the use of a shared resource incurrs additional overhead - the channel program must be created and managed by a single XJS process (again, the "Mazda Acess Manager"), which must communicate with the appropriate device Access Managers.

The reporting of modem signal changes (DCD, CTS, etc.) is a good example of a situation in which this facility would be used.

In order to implement this facility, the "Mazda Access Manager" would create a channel program structured as of a queue of "read immediate" Channel Command words. A Wankel task which wanted to report an event would write an ID, and status information into the address field of the current Channel Command word of this channel program, and terminate it in the normal manner. When the Mazda Access Manager receives the interrupt corresponding to this Channel Command, it would use the ID byte to identify the process to which it would send a message, reporting the event.

[Proposed Channel Command format to be supplied...]

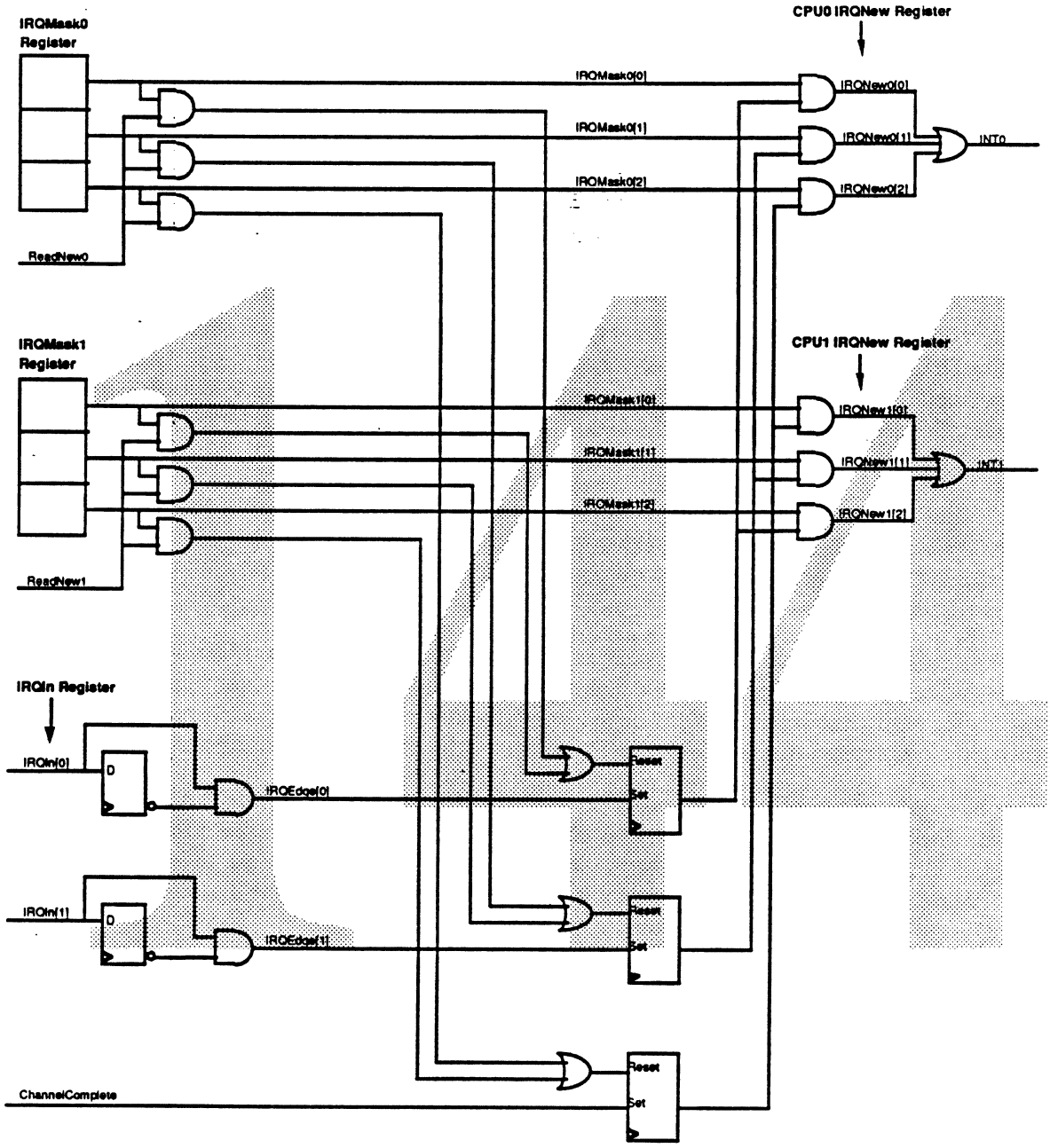
Interrupts

Mazda will provide a centralized interrupt reporting facility for Jaguar. The two sources of these interrupts are internal Channel Command completions from each executing channel program, and external interrupt signals received from other devices on the motherboard as well as boards in the expansion slots. An interrupt output is generated for each of the XJS processors.

A Channel Command completion interrupt will be generated whenever a Channel Command which has its Interrupt bit set finishes execution. This will cause an interrupt request flip-flop to be set. The only action required of the ISR which services this interrupt is to send an IPC message which will activate the appropriate Access Manager. Upon receiving this message, the Access Manager can scan the Ready bits of the Channel Commands in its channel program to determine which one(s) have completed.

External interrupts are level sensitive inputs to Mazda. The approach taken is to set an interrupt request flip-flop when an asserting transition is detected on these lines.

The following diagram illustrates the interrupt logic. Two external interrupts (IRQIn0, IRQIn1), and one channel completion interrupt are depicted. In fact, there will be several external interrupts, and a large number of Channel Command completion interrupts. The following section will explain in detail the functions performed by the interrupt logic.



Mazda Interrupt Structure

Programming Model, and Multiprocessor Considerations

Jaguar has many sources of interrupts, including a large number from channel programs, in addition to BLT slots, Video "Beam Chasing", etc. Since we are also designing a 2-processor system, the question of how to "vector" the interrupts between the processors arises.

Many concerns have to be addressed in the interrupt model for the Jaguar:

To which processor does a particular interrupt get directed?

If two processors can "get" a given interrupt, how do we make sure that interrupt is not processed twice?

How does the system specify which processor allocated to a given interrupt?

The following model addresses these issues.

The Programming Model

(Note that for the following discussion, we are assuming that the external interrupts remain asserted until some software has done some explicit action (e.g., "tickling" some magic register in a device. Interrupts will be asserted once, stay asserted and become de-asserted only when an interrupt service routine has executed some code. This seems to be a reasonable model for the external interrupts which we are likely to see in Jaguar.)

Mazda contains 1 register (**IRQIN**) which always contains the state of the external interrupts currently pending. Values in this register are updated to reflect the "raw" signals at the "pins" (suitably synchronized). This register can be read at any time, by anybody. (Channel Command completion interrupts are not represented in this register).

For each processor (**n**), two additional registers are observable. One of these, the **IRQMASKn** register is a read/writable (real) register which holds the interrupt mask bits for the processor. Any interrupt whose corresponding bit is clear in this register will be ignored for the purposes of initiating an interrupt for that processor (i.e., it prevents that processor's **INT*** line from being asserted for that interrupt).

Another processor-unique (virtual, read-only) "register" (**IRQNEWn**) is available which reflects "new" interrupts. The bits in this register are the AND of the edge-detected interrupts and the **IRQMASKn** register. Edge-detection here means that a bit is potentially set only when the corresponding bit in **IRQIN** goes from a 0 to a 1. When any bit in an **IRQNEWn** register is set, the corresponding **INT*** line is asserted; i.e., **INT*** (asserted) is the OR of all bits in **IRQNEWn**.

An interesting property of the **IRQNEWn** registers is that all of their bits are cleared whenever the register is read. By virtue of the edge-detection on its inputs, any bit which was set and then cleared by a read of IRQNEWn will remain clear until the corresponding **IRQIN** bit goes back to a 0 then to a 1; hence, the name "new". The net result of these mechanisms is that an interrupt will only be generated on assertion transitions of the interrupts; not continuously just because an interrupt line (i.e., a bit in **IRQIN**) stays asserted.

Ignoring for the moment a multi-processor system, the interrupt service routine simply reads its **IRQNEWn** register and uses the bits to "dispatch" Interrupt Service Routines (ISRs). Assuming that no new interrupts come in for a while, the **INT*** line is cleared immediately upon reading the **IRQNEWn** register. This could allow ISRs to be run at non-interrupt level, since **INT*** will not be held until all of the ISRs have done their thing (as it is in today's Mac world). This is because the state of **INT*** is not predicated upon active interrupt lines, but rather their edge-detected states.

The interrupt handler can read **IRQNEWn** again, just before exit, to determine if any new interrupts have come in since its initial entry. When the **IRQNEWn** is clear, the interrupt handler can exit (or, initiate a dispatch), enabling interrupts. Assuming that ISRs have a special high-priority dispatch mechanism, they would probably be run at this time. The important point to re-state is that there is no logical problem in running the ISRs with interrupts enabled since their initiating event (new interrupt seen) has been "cleared".

Now, for the case of multi-processors. There are two cases to consider. The first is where the **IRQMASKn** registers have been set so that the interrupts are mutually exclusive between the CPUs; i.e., the AND of all IRQMASKs is 0. This is really not much different than the simple case, since the same interrupt will never be signalled to both processors.

The second case is more interesting. In this case, some (or all) interrupts can be processed by either CPU; i.e., their **IRQMASKn** registers have bits in common. In this case, we would (probably) like the CPU which can process the interrupts first to handle all the interrupts. At least, we would like interrupts which are going to be handled by one processor to be prevented from being seen by the second CPU.

The way this is handled is as follows: The individual **IRQNEWn** registers on not real. Rather, they are derived, when read, from the data in a single (real) **IRQNEW** register and the CPU's **IRQMASKn** register. The hardware clears all bits in **IRQNEW** which are presented to (i.e., read by) a processor when accessing its **IRQNEWn** "register".

Suppose that all interrupts are directed to both CPUs; i.e., their **IRQMASKn** registers are all ones. The interrupt handler is the same as above, except that it has a special escape hatch up front. If the first read of its **IRQNEWn** register is all clear, then the interrupt handler simply returns immediately. This would happen if both processors get interrupted, but one of them reads its **IRQNEWn** first. Since all bits in its **IRQMASKn** are set, all bits in **IRQNEW** are cleared. Thus, when the second CPU reads its **IRQNEWn**, it will find all the bits clear. Since it has no work to do, it simply exits.

If the second CPU reads a bit in **IRQNEWn**, it is not being handle by the first, since it must have shown up after the first CPU read its **IRQNEWn**. The second CPU would then handle this new interrupt, while the first CPU is handling the originals.

The argument for the above scheme is that it allows a fully symmetric interrupt model (i.e., both processors can process any interrupt) in a clean fashion. Of course, it handles the opposite case (interrupts specialized to CPUs) , as well as anything in between.

External Interrupts

As mentioned above, each of the expansion cards generates an interrupt line that is received by Mazda. As for all interrupts, the Interrupt Mask registers on Mazda contain a bit for each of these interrupts that controls whether the interrupt is to be passed on to the processor or not.

In addition, there is a provision for each of these interrupt lines to cause a memory-mapped write to another expansion card. This allows card-to-card communication even if the initiating card is not a bus master.

The basic idea is that hardware within Mazda is pre-conditioned so that when an interrupt line is asserted, the hardware executes a write to an address within the "destination" card. The data that is written is always 0's; this eliminates the need for defining a source of the write.

The mechanism used to accomplish this is a very simple Channel Program. It would consist of the following two Channel Commands:

Loop:

    Wait for interrupt, then write 0's to this address

Jump to Loop

Since it is possible for there to be an active Channel Program for each of the interrupt lines, the Channel Command File in Mazda must have three locations reserved for the Wankel task that controls this operation. When the task interprets the first Channel Command above, it arms a hardware module to start looking for the appropriate interrupt line to be asserted. When it is asserted, the hardware module sends a "write 0's" request to the Bus Interface Unit. The BIU reads the proper address from the Channel Command file and carries out the write operation. The task then executes the jump, looping back to the first Channel Command. The task will continue to execute the Channel Program until it is aborted by the processor.

The expansion card that is the destination of the write is responsible for taking whatever steps are necessary to clear the interrupt line.

Section 3

Mazda I/O Controller

Wankel Processor

Wankel is a simple RISC processor that has been optimized for fast context switching between tasks. The code for Wankel consists of a set of tasks (one for each active Channel Program). Each task is responsible for interpreting Channel Commands, initiating DMA transfers by its associated I/O Module, monitoring the progress of those DMA transfers, implementing the specific protocol of its associated I/O device(s), and interfacing with the Bus Interface Unit to fetch subsequent Channel Commands.

Despite the fact that there is much for a task to do, it will spend most of its time waiting: for a DMA transfer to complete, for a Command Word to be fetched from memory, for a requested register value to be returned, for a Channel Program that will give it something to do in the first place. When the task is truly executing, the types of operations that it must perform are relatively simple (mostly masking and testing along with loads and stores). Therfore, Wankel has a significant portion of its silicon dedicated to optimizing the "waiting" part of a task, and relatively little used for the traditional CPU blocks like ALU's and register files.

The description of Wankel that follows will start with a brief overview of its architecture. Next will be a description of the Wankel instruction set. This will be followed by a description of operation of the processor, with special emphasis on the memories that support the fast task switch capability. Next will be a description of Wankel's own task0 and some of the operations that it must perform. The final sub-section deals with Wankel's interfaces to the Bus Interface Unit and the I/O Modules, and will present some Wankel code examples.

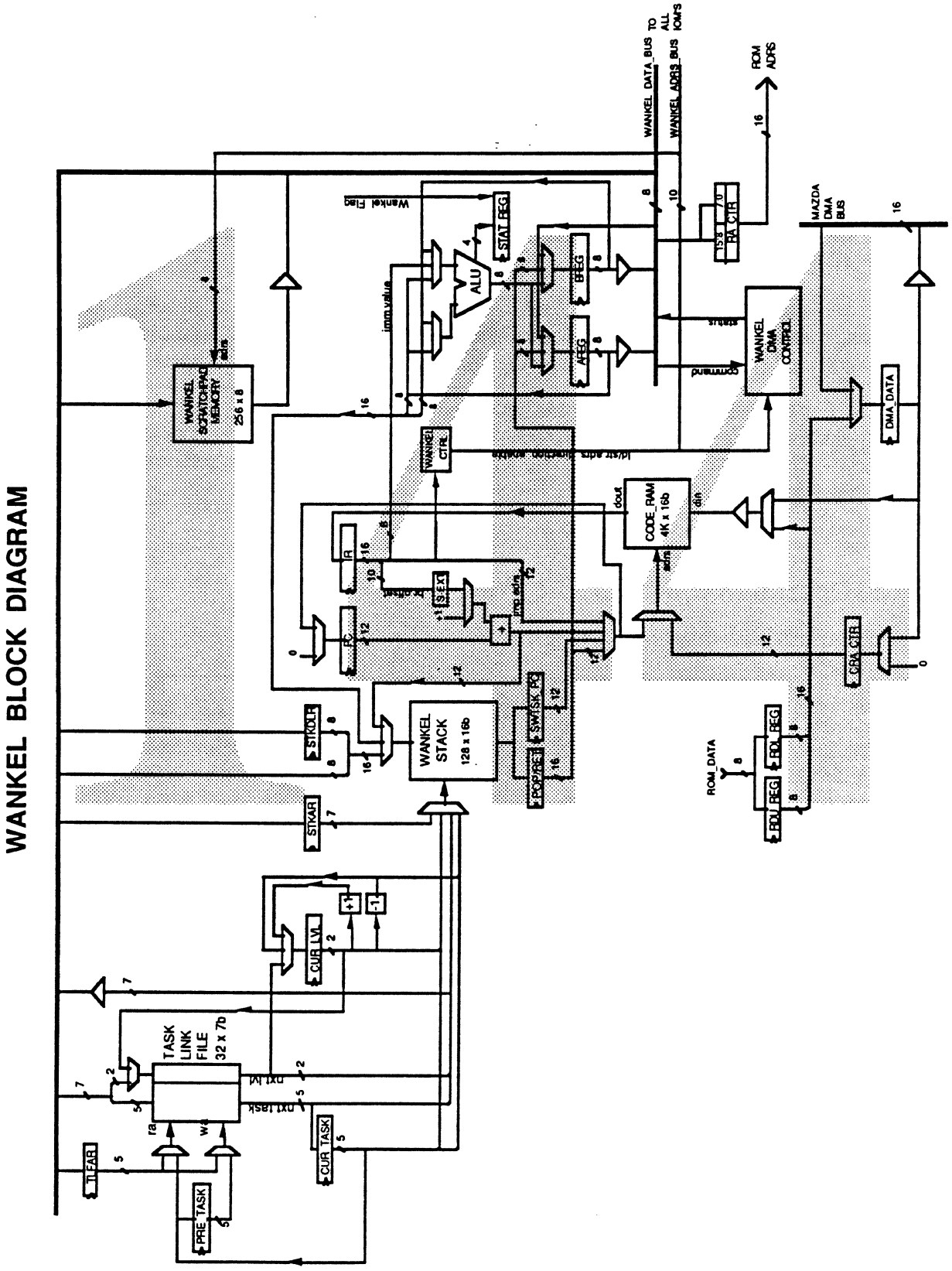
Architecture

Wankel is an 8-bit data, 16-bit instruction CPU. The Code RAM that contains all of its code is 4K entries deep (8K bytes). A register-level block diagram of Wankel is on the following page. The names of registers appearing on the block diagram will be used throughout this section of the ERS.

Wankel's computation unit is organized around an ALU and two accumulator-style registers, AREG and BREG. A variety of arithmetic/logical operations may be performed using AREG and BREG as inputs or using one of the registers with an immediate value from the instruction. Wankel has four status flags that are updated by the result of every ALU operation.

Loads and stores are performed across the Wankel Data Bus using either AREG or BREG as source/destination. In addition, the architecture provides a capability of directly loading a status flag, where it is immediately available for a conditional branch instruction. Memory addresses are 10 bits on Wankel. The architecture provides two methods of generating memory addresses; either directly from an Address field in the instruction or by concatenating 2 bits of the Address field with the contents of one of the registers. The architecture provides a number of combinations of the status flags for conditional

branches.


WANKEL BLOCK DIAGRAM
WANKEL DATA BUS TO ALL IOVS
WANKEL ADRS BUS
ROM ADRS
MAZDA DMA BUS
WANKEL SCRATCHPAD MEMORY 256 x 8
ALU
STAT REG
BREG
AREG
WANKEL DMA CONTROL
DMA_DATA
WANKEL CTRL
CODE_RAM 4K x 16b
IR
PC
SEX
WANKEL STACK 128 x 16b
STKDH
STKAR
SWISS PC
PCPIPE
CUR_LVL
TASK LINK FILE 32 x 7b
CUR_TASK
PRE_TASK
TLEAR
CBA_CTR
ROM_DATA
ROM_REG

The architecture provides a 4-entry, 16-bit stack for each task executing on Wankel. The stack may be used by a task for subroutine return addresses and/or register saving. In addition, the hardware uses the stack as a place to remember the task's "return PC" when it switches context. The hardware utilizes another memory, the Task Link File, to aid it in linking from task to task. This file is generally not visible to individual task code, but Wankel's task0 has access to it when adding or removing tasks.

Finally there is a DMA controller associated with Wankel that works in concert with task0 in providing data transfers to and from Wankel itself.

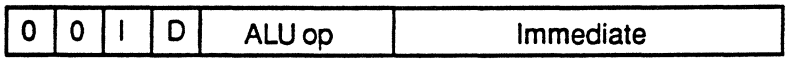
Instruction Set

The Wankel instruction set is composed of the following groups: instructions that involve an ALU operation, loads, stores, branches, jumps, and a group creatively labelled "other instructions". The last group consists primarily of instructions that manipulate the Wankel Stack and Task Link File.

Except for one instruction (wrcod), all instructions are designed to execute in a single Wankel clock cycle (40 ns). However, there are certain code sequences involving Wankel Stack operations that will produce one or two cycle pipeline delays. These code sequences are detailed in the sub-section titled Wankel Stack later in this section.

ALU Operations

Wankel instructions involving the ALU have the following format:



The destination for the result of an ALU operation is selected by the D bit of the instruction. If the D bit is 0, the destination is AREG. If the bit is 1, the destination is BREG. The destination register is also one of the inputs to the ALU operation.

The second input to the ALU operation (call it the source) is selected by the I bit of the instruction. If the I bit is 0, the source is the register that is not the destination (i.e., if AREG is the destination, then BREG is the source, and vice versa). If the I bit is 1, the 8-bit Immediate field of the instruction becomes the source operand.

The 4-bit ALU op field of the instruction defines the operation that is to be performed on the source and destination operands. The following table lists the instruction mnemonic and operation that occurs for each value of the ALU op field.

0	—	not used
1	and	Dest <— Dest 'and' Source
2	or	Dest <— Dest 'or' Source
3	xor	Dest <— Dest 'xor' Source
4	add	Dest <— Dest + Source
5	addc	Dest <— Dest + Source + CarryFlag
6	sub	Dest <— Dest - Source

7	subb	Dest <--- Dest - Source - CarryFlag
8	rsb	Dest <--- Source - Dest
9	rsbb	Dest <--- Source - Dest - CarryFlag
A	test	Dest 'and' Source (no change to Dest)
B	cmp	Dest - Source (no change to Dest)
C	mov	Dest <--- Source
D	shl	Dest <--- Dest shifted left by 'Source' bits, filling with zero
E	shlc	Dest <--- Dest shifted left by 'Source' bits, filling with CarryFlag
F	rot	Dest <--- Dest rotated left by 'Source' bit

Four of the status flags (Carry, Overflow, Negative, and Zero) are updated as a result of an ALU operation.

Loads and Stores

Wankel communicates with the I/O Modules, the Bus Interface Unit, and internal Wankel registers through memory-mapped loads and stores. The Wankel Address Bus is 10 bits wide, allowing 1024 memory-mapped locations to be defined. The first 256 addresses are reserved for reads and writes of the Wankel Scratchpad Memory.

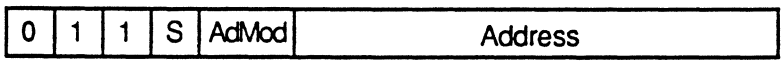
Load instructions on Wankel have the following format:

The destination of a load is selected by the D bit of the instruction. If the D bit is 0, the destination is AREG. If the D bit is 1, the destination is BREG.

The method of generating the address for the load is selected by the 2-bit AdMod field of the instruction. If AdMod equals 0 or 1, the 10-bit Address field of the instruction is used directly. If AdMod equals 2, the address is formed by concatenating the most-significant two bits of the Address field with AREG. If AdMod equals 3, the address is formed by concatenating the most-significant two bits of the Address field with BREG.

The four ALU status flags (Carry, Overflow, Negative, and Zero) are not updated by a load instruction. The Wankel Flag is always updated by a load instruction. When Wankel loads data from a memory-mapped address, that address may have a status bit associated with it. This bit acts like a ninth bit of the Wankel Data Bus that is clocked into the Wankel Flag. It is then available for a conditional branch on the very next instruction without further masking and testing. Some of the uses of this flag would be to indicate the validity of the data being read or to indicate the readiness of the addressed unit to receive a command.

Store instructions on Wankel have the following format:



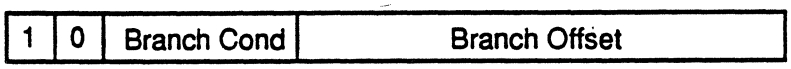
The source for the store data is selected by the S bit of the instruction. If the S bit is 0, the source is AREG. If the S bit is 1, the source is BREG.

The AdMod and Address fields have the same meaning as for load instructions.

None of the status flags, including the Wankel Flag, are updated by a store instruction.

Branches

Branch instructions on Wankel have the following format:



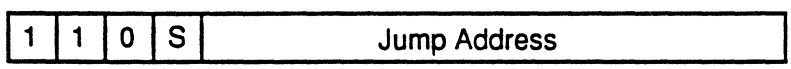
The 10-bit Branch Offset field is sign-extended to 12 bits and then added to the PC address of the branch instruction to form the target address in the Wankel Code RAM.

The 4-bit Branch Cond field specifies a combination of status flags to use in determining if the branch is taken. The following table lists the instruction mnemonic and the equation for determining branch taken for each possible Branch Cond value. In the table, the following letters are used to represent the five status flags: Carry(C), Overflow(V), Negative(N), Zero(Z), and Wankel(W).

0	nop	no operation (unconditional branch not taken)
1	br	unconditional branch taken
2	bc	C
3	bnc	~C
4	bo	V
5	bno	~V
6	bz/beq	Z
7	bnz/bne	~Z
8	bneg	N
9	bpos	~N
A	bw	W
B	bnw	~W
C	blt	N * ~V + ~N * V
D	ble	Z + N * ~V + ~N * V
E	bgt	~(Z + N * ~V + ~N * V)
F	bge	~(N * ~V + ~N * V)

Jumps

Jump instructions on Wankel have the following format:



If the S bit is 0, the instruction is a normal jump. If the S bit is 1, the instruction is a "jump to subroutine" (jsr). In either case, the Jump Address field represents the location in the Wankel Code RAM where execution is to continue.

Execution of a jsr instruction consists of pushing PC+1 onto the Wankel stack and then jumping to Jump Address. There is a more detailed description of Wankel stack operation later in this section.

Other Instructions

The remaining Wankel instructions have the following format:

The 3-bit OpCode field defines the instruction according to the following table.

0	**push**	push AREG and BREG to stack
1	**pop**	pop AREG and BREG from stack
2	**ret**	return from subroutine
3	—	not used
4	**swtsk**	switch to the next Wankel task
5	—	not used
6	**wrcod**	write Wankel Code RAM
7	—	not used

The **push** instruction pushes AREG/BREG onto the Wankel Stack for later retrieval by a **pop** instruction.

The **pop** instruction pops the Wankel Stack and loads the value that was popped into AREG/BREG.

The **ret** instruction is the normal counterpart to the **jsr** instruction. Execution of a **ret** instruction consists of popping the Wankel stack, loading the value that was popped into PC, and continuing execution from there.

The **swtsk** instruction is executed by a Wankel task when it has determined that it has nothing to do. This occurs whenever the task must wait for some requested action to complete. Examples of this are waiting for a DMA transfer to complete or waiting for the next Channel Command in a Channel Program to be fetched.

It may be necessary for a Wankel task to insert a **swtsk** instruction even when it has something to do. This will become necessary if the latency of a round trip around the task loop with each task executing its worst-case stretch of code could cause some data to get dropped. More investigation is required here to determine that worst-case latency.

The **swtsk** instruction performs the following actions:

1. Stores PC+1 to the Wankel Stack at the current task and stack level so it can be retrieved the next time the exiting task runs. Note that the following registers are not saved on a context switch: AREG, BREG, and the Status Flags. If the task will need any of these values when it next executes, it must take explicit action to save them before executing the **swtsk** (such as executing a **push** or storing them to the Scratchpad).
2. Stores the current task level to the Task Link File (at the address of the task that called the exiting task) so it can be retrieved the next time the exiting task runs.
3. Fetches the task number and stack level for the incoming task from the Task Link File.
4. Uses the information from step 3 to fetch the incoming task's Program Counter from the Wankel Stack and loads it into PC. Execution within the new task's environment then begins at that address.

There is a more detailed description of Wankel Stack operation and Task Link File operation later in this section.

The **wrcod** instruction is used to suspend Wankel code execution for one clock cycle while Wankel's Code RAM is loaded with a line of code. This instruction is executed periodically by Wankel task0 when it has initiated a DMA transfer from memory to the Wankel Code RAM. The **wrcod** instruction takes 2 cycles to execute. During the first cycle, the Wankel DMA Controller assumes control of the Code RAM and loads a line of code if it has one available. During the second cycle of the **wrcod**, control of the Code RAM reverts back to Wankel to allow normal accessing of its next instruction.

Refer to the sub-section titled "Adding a Task to Wankel" later in this section for a description of the process of downloading Wankel code.

Description of Operation

The Wankel processor has only two pipeline steps: instruction fetch and execution. While it is executing one instruction, it is fetching the next instruction for execution from its Code RAM. This is true even when it is executing an instruction that changes program flow (**br**, **jump**, **jsr**, **ret**) or when it is executing an instruction that changes context (**swtsk**).

Operation of the execution phase is straightforward and has been adequately described in the Instruction Set subsection. The instruction fetch phase is the more interesting part of Wankel and the remainder of this subsection will describe the operation of its major blocks.

Program Counter

The Program Counter (PC) always contains the Code RAM address of the instruction that is currently executing (IR). The address of the next instruction to be executed (and thus the next value of PC) comes from one of five possible sources:

1. If the current instruction is a **jump** or **jsr**, the next PC value is the Jump Address field of the instruction (IR[11:0]).

2. If the current instruction is a branch taken, the next PC value is obtained by adding PC to the sign-extended Branch Offset field of the instruction (IR[9:0]).

3. If the current instruction is a **ret**, the next PC value is obtained from the RET/POP register. The purpose of the RET/POP register is to accelerate the return address so a stack access is not necessary when a **ret** is executed. The following section on Wankel Stack operation contains a description of how this register gets loaded.

4. If the current instruction is a **swtsk**, the next PC value is obtained from the SWTSK_PC register. The purpose of the SWTSK_PC register is to accelerate the next task's starting PC address so a stack access is not necessary when a **swtsk** is executed. The following section on Wankel Stack operation contains a description of how this register gets loaded.

5. If the current instruction is a branch not taken or an instruction that does not alter program flow, the next PC value is obtained by adding 1 to PC.

Wankel Stack

The Wankel Stack contains a location for holding a PC value or a saved AREG/BREG pair for each stack level of each Wankel task. The structure of the Wankel Stack is actually one of 32 separate 4-entry stacks. This limits Wankel to having 32 tasks, each of which may have four stack levels. Addresses to the stack are constructed by concatenating a 5-bit task number with a 2-bit stack level.

Associated with the Wankel Stack are two accelerator registers: POP/RET and SWTSK_PC. The hardware always tries to keep the contents of these two registers valid. The POP/RET register is valid when it has been loaded with the stack entry that is currently the top of the stack. The SWTSK_PC register is valid when it has been loaded with the starting PC address of the next task to execute. Without these two registers, execution of a **pop**, **ret**, or **swtsk** would require a Wankel Stack access and prevent these instructions from executing in a single cycle. With these registers, the hardware works "behind the scenes" during the execution of other instructions to set up for a **pop**, **ret**, or **swtsk**.

The contents of the POP/RET register are invalidated by a **push**, **pop**, **jsr**, **ret**, or **swtsk** instruction. The **swtsk** instruction also invalidates the contents of the SWTSK_PC register. The hardware can restore the validity of one of these registers every cycle provided that the instruction being executed in that cycle does not require access to the Wankel Stack. The result of this is that there are certain sequences of code that will cause a 1-cycle pipeline delay. Most of the sequences are unrealistic combinations of instructions (like following a **jsr** with an immediate **ret**). However, the following sequences of instructions are realistic and should be noted: (1) a **ret** followed by another **ret**; (2) a **ret** followed by a **pop**; (3) a **pop** followed by a **ret**.

Normally a task would have no need for writing the Wankel Stack explicitly under program control. However, task0 must have this capability when adding new tasks. Explicit writes to the stack are accomplished through a series of memory-mapped stores: to the Stack Address Register (STKAR) to set up the address to be written; to the Stack Data Lower Register (STKDLR) to transfer the least-significant byte

of the data; and finally to the stack itself (STK) accompanied by the most-significant byte of the data to be written.

Task Link File

The Task Link File contains the information necessary for Wankel to link to the next task when a **swtsk** instruction is executed. The size of the Task Link File is 32 entries by 7 bits. An entry in the file consists of a 5-bit NXT_TASK field and a 2-bit NXT_LVL field. NXT_TASK is the number of the next task to execute after the current task. NXT_LVL is the stack level that the next task was executing at the last time it ran.

During normal task operation, the Task Link File is used only when a **swtsk** instruction is executed. The first action is to store the current task's stack level to the file. This is written at the address of the *previous* task (PRE_TASK) so that it is available the next time around the task loop when Wankel is about to switch to this task again. The next action on a **swtsk** is to clock the NXT_TASK and NXT_LVL fields into CUR_TASK and CUR_LVL, the registers keeping current task number and current stack level.

Like the Wankel Stack, task0 needs direct read and write access to the Task Link File when adding or removing tasks. This is accomplished by writing an address to the Task Link File Address Register (TLFAR), followed by a read or write of the Task Link File itself (TLF).

Task0

Task0 is a special task dedicated to DMA transfers and bookkeeping operations involving Wankel itself. Like the other tasks that execute on Wankel, it has a dedicated DMA Controller that is used for transferring blocks of data. Starting at initialization, task0 is always executing on Wankel, taking its turn in the task loop. Except for the XJS initialization process, task0 executes Channel Programs created by XJS just like any other Wankel task. The remainder of this subsection describes the operations performed by task0.

Initialization

Mazda controls the reset lines to the XJS processors. There are 3 pins on Mazda associated with reset: an input pin SYS_RESET and two output pins XJS0_RESET and XJS1_RESET. When a hardware reset is issued (SYS_RESET is asserted), Mazda immediately asserts both XJS0_RESET and XJS1_RESET. Mazda continues to assert both lines after SYS_RESET has been de-asserted.

There are two flags associated with initialization in the memory map of Mazda space. The first flag is MEM_READY, which exists at memory address <TBD>. MEM_READY is used to control the download of XJS boot code. The second flag is BOOT_XJS1, which exists at memory address <TBD>. When SYS_RESET is asserted, Mazda clears both of these bits. When the processor issues a write to the memory-mapped address of one of these flags (the write data is ignored), the flag is set. Mazda's use of these flags is described below.

On reset, Wankel is disabled from executing instructions. A hardware state machine downloads Wankel's task0 code from off-chip ROM to the Code RAM starting at address 0. Wankel's PC is initialized to 0.

Initialization of the Task Link File consists of writing 0's to location 0. This makes task0 link back to itself when it executes a **swtsk** instruction.

After performing the above actions, the reset state machine turns control over to Wankel and it begins executing its task0 code.

The first action that task0 must perform is to download the XJS boot code from ROM to system memory. The download must be accomplished in two stages, with the first 4K bytes getting downloaded immediately, with the remainder to be downloaded after the processor has configured memory. The reason for downloading only the first 4K bytes is that, on reset, the System Controller chip can guarantee that only the block of memory from address 0 to address 4K-1 is available. Furthermore, addresses to memory other than this block are not guaranteed to remain the same when the actual memory configuration is realized.

To perform the download, task0 must create a Channel Command with the correct Address and Count values and write it into the Channel Command File in the Bus Interface Unit. Once the Channel Command is there, Wankel's DMA Controller starts the process of transferring the boot data from ROM to memory starting at address 0. When the DMA Controller reaches the 4K byte boundary, it suspends the transfer. Task0 is notified, and a command is issued to de-assert XJS0_RESET (appropriately synchronized to the system clock). The reset line to the other processor, XJS1_RESET, remains asserted.

XJS0 is designated as the master processor. After XJS0_RESET has been de-asserted, XJS0 begins executing the boot code at address 0. When XJS0 is ready for the slave processor to come online, it replaces the instruction at address 0 with a branch to the start of the slave processor boot code. XJS0 then issues a write to the BOOT_XJS1 address in Mazda. This causes Mazda to de-assert XJS1_RESET. XJS1 will come out of reset and start executing at address 0, where it will immediately branch to the slave processor boot code.

At some point, XJS0 will test memory and set the configuration registers accordingly. After this occurs, it is safe for Mazda to download the remainder of the XJS boot code. XJS signals its readiness by issuing a write to the MEM_READY address in Mazda. When the Wankel DMA Controller sees this write, it continues the download process. The processor waits for the last memory location in the boot image to get written. When it sees the proper value appear there, it knows that the download is complete.

The XJS code that is downloaded from ROM is not sufficient for a complete boot. The ROM code merely allows XJS to perform some diagnostic/initialization operations and then set up the appropriate Channel Programs to obtain the boot image from the boot device. Normally the boot device will be the hard disk, but software is free to define overrides to this. These overrides may be either automatic (presence of a disk in the floppy drive) or user-selected (stored in the Parameter RAM).

Adding a Task to Wankel

Wankel provides the capability to dynamically add and remove tasks without stopping Wankel instruction execution. This feature will become important if the total amount of Wankel code exceeds the amount of Code RAM that can be placed on Mazda, and it therefore becomes necessary to swap code and/or tasks. It should be noted, however, that this will be a slow process. Clearly more investigation is needed to determine the total amount of Wankel code. In any case, this feature provides software with the basic mechanism for initializing the Code RAM, Task Link File, and Wankel Stack.

To add a task to Wankel, XJS creates a Channel Program for Wankel's task0 to execute. This Channel Program would consist of the following Channel Commands:

1. Load starting PC address for writing to Code RAM
2. Write block of Code RAM
3. Add task

Each of these CC's would have the same format as CC's for the I/O Modules, with an Address and Count field for the transfer.

After XJS has created the Channel Program, it notifies task0 of its existence by writing the pointer to the start of the Channel Program to a memory-mapped register in Mazda. The following is a description of the specific actions that occur on each CC.

CC 1: Load starting PC address for writing to Code RAM

Task0 decodes the first CC and issues a write to the command register in Wankel DMA Control. The command requests a DMA transfer from memory to the Code RAM Address Counter (labelled CRA_CTR in the Wankel block diagram). When DMA Control has completed the transfer, it sets a status bit that task0 reads and knows it may move on to the next CC.

CC 2: Write block of Code RAM

Task0 issues a write to the command register in Wankel DMA Control, requesting a DMA transfer from memory to Wankel Code RAM. DMA Control then starts reading the block of data from memory. However, DMA Control can not write into the Code RAM whenever it receives data from memory, since Wankel is executing out of the Code RAM. This is the reason for the **wrcod** instruction described in the Instruction Set section. When task0 executes a **wrcod** instruction, DMA Control may take over the address to the Code RAM for one cycle and write to it.

CC 3: Add task

The Address and Count fields of this CC specify a block of memory that contains the number of the task to be added and the Wankel PC address where the task begins execution. Wankel DMA Control reads the data from memory and feeds it to task0. Task0 code then makes the appropriate memory-mapped reads and writes of the Task Link File necessary to place the new task in the linked list of tasks. Task0 also writes the appropriate Wankel Stack address with the new task's starting PC. This completes the preparation for the new task, and it will execute during the next pass around the task loop.

Removing a Task from Wankel

To remove a task from Wankel, XJS creates a Channel Program for Wankel's task0 to execute. This would consist of a single CC with a Command field of "Remove task", and with Address and Count fields pointing to a byte in memory that contains the number of the task to be removed. Wankel DMA Control reads the byte from memory and feeds it to task0. Task0 code then makes the appropriate memory-mapped reads and writes of the Task Link File necessary to remove the indicated task from the linked list of tasks.

Wankel Interfaces

As described earlier, Wankel communicates with the Bus Interface Unit and all the I/O Modules through memory-mapped loads and stores. Each unit that interfaces to the Wankel Data Bus and the Wankel Address Bus will have a certain number of byte-wide registers defined that Wankel may read or write directly. Each of these registers will have a location assigned to it within Wankel's memory-mapped address space. Any registers that interface directly with Wankel must be able to satisfy the requirement of single-cycle loads and stores. These registers are called direct-access registers.

There are other registers that Wankel must communicate with indirectly. Registers fall within this category when they are not able to receive or transfer data in a single cycle. An example of this type are registers within the I/O chips. A sequence of two load/store instructions is required to read or write these indirect-access registers.

To write an indirect-access register, Wankel must first issue a store to a direct-access register within the same module. Typically this would be called the Data Register, and the data stored into it would be the data that Wankel wanted to eventually end up in the target indirect-access register. Next Wankel would issue a store to a different direct-access register within the same module. Typically this would be called the Command Register, and the data stored into it would include a Command field and an Address field. The decode of Command would tell the module that it is to take the data currently in the Data Register and transfer it to the indirect-access register selected by the Address field.

To read an indirect-access register, Wankel would first issue a store to the Command Register, requesting a read of the indicated indirect-access register. At some later time, Wankel would read the contents of the Data Register, along with the Wankel Flag that indicates if the data being read is valid. Normally the time between the store that makes the read request and the actual load of the data would be a complete cycle around the task loop, since the task should execute a **swtsk** instruction while waiting. However, there may be certain critical tasks that need to wait for the data to be returned.

CC File

To illustrate the interaction between a Wankel task and its associated DMA module, included here is the definition of the Wankel / CC File interface. Also included is the Wankel code used to obtain a CC from the CC File.

A description of the operation of the CC File is given in the following section on the Bus Interface Unit. For the purposes of this example, we will assume that the Bus Interface Unit is in the process of fetching a new CC for the task and that the task is currently executing a "test and switch task" loop waiting for a new CC. Another assumption is that this task has been assigned CC File address 35 as the location for its CC's.

The following table lists all the memory-mapped operations associated with reading the CC File.

Identifier	R/W	Action
CCrdy<n>	R	Enables CCrdy[8n+7 : 8n] onto Wankel Data Bus
CCreq	R	Enables CCbusy flag onto Wankel Flag Bus
CCreq	W	Initiates CC File read at address given by Wankel Data Bus bits 5:0; sets CCbusy flag; clears CCrdy bit selected by Wankel Data Bus bits 5:0; clears CCack flag

CCdata<n>	R	Enables CCdata[8n+7 : 8n] onto Wankel Data Bus; enables CCack flag onto Wankel Flag Bus; clears CCbusy flag

Here is a description of some of the terms used in the above table:

CCrdy is a 64-bit register that contains a bit for each entry in the CC File. If set, the bit indicates that the corresponding CC File address has been loaded with a new CC. The register is divided into eight 8-bit registers (CCrdy0-7) that may be read directly by Wankel.

CCbusy is a flag that is set by the CC File control logic when Wankel requests a read from the CC File. It is cleared when the requested data is read by Wankel. This bit is required because the CC File is indirect-access by Wankel, and an interlock is required to prevent one Wankel task from clobbering the request of another task.

CCreq is a location that a task reads when it wants to get the status of CCbusy and it writes to when it wants to initiate the read of a CC File entry.

CCack is a flag that indicates if the CCdata register contains valid data. When a task reads CCdata, it receives this flag on the Wankel Flag Bus.

CCdata is a 64-bit register that is loaded with the requested CC when it is read from the CC File. The register is divided into eight 8-bit registers (CCdata0-7) that may be read directly by Wankel.

The Wankel code required to implement the fetch of a CC is shown below. The code is composed of two segments, a task-specific part that must be replicated for each Wankel task and a common subroutine that all tasks execute. It is believed (hoped?) that much of the code for Wankel can be turned into common subroutines.

Individual task code

```
CCrdy_loop:
        swtsk
Start:
        ld      a,CCrdy4        ; load 8-bit register containing my CCrdy bit
        test    a,8             ; mask off all but my bit
        bz      CCrdy_loop      ; continue if CC is ready; else loop
        mov     b,35            ; BREG <--- address of my CC file entry
        jsr     GetCC
```

Subroutine common to all tasks

```
GetCC:                          ; common subroutine for reading CC Cmd byte
        ld      a,CCreq         ; load Wankel Flag with CCbusy flag
        bnw     ReadCC          ; continue if not busy; else try again
        push                    ; save BREG (address of CC file entry)
CCbusy_loop:
        swtsk
        ld      a,CCbusy        ; load Wankel Flag with CCbusy flag
```

```
        bw      CCbusy_loop         ; continue if not busy; else loop
        pop                         ; BREG once again has CC file address
ReadCC:
        st      b,CCreq             ; start the read
CCdata_loop:
        swtsk
        ld      a,CCdata0           ; load Cmd byte to AREG and CCack to Wankel Flag
        bnw     CCdata_loop         ; continue if data ready; else loop
        ret                         ; return; task may read more bytes from CCdata
```

Bus Interface Unit

BIU Interfaces

The bus operations of the Mazda I/O chip are controlled by three main sections of logic. Bus operations can be initiated by XJS, Wankel, or any of the individual DMA channels. XJS may request the BIU to start executing a Channel Program by writing channel program pointers to the CP pointer file in Mazda. Wankel may request the BIU to fetch Channel Commands (CC's) by setting bits in the "Wreq" register. And DMA channels may request the movement of data between main memory and a device buffer by setting a bit in the "Dreq" register. Each of these sources of control will be discussed in detail below.

XJS/BIU Interaction

As discussed elsewhere in this document, I/O control in the Jaguar system(s) is governed by "Channel Programs". The execution of a channel program by Mazda is initiated by the host processor (XJS). The "request" for Mazda service is made by XJS writing a Channel Program Pointer (CPP) to the CPP file in the BIU. The CPP file is an area of RAM in the BIU that maintains the addresses of all of the Channel Commands present in the CC file. The act of XJS writing a pointer to the CPP file causes a bit to be set in the "Xreq" register. This will in turn cause a request for a fetch of the CC at the pointer address. When the CC is fetched and written into the CC file the corresponding bit is cleared in the Xreq register.

Wankel/BIU Interaction

Wankel too can request bus activity from the BIU. This typically happens when the need arises to fetch the next CC in a channel program. A Wankel request to the BIU is made by setting a bit in the "Wreq" control register. A bit set in this register will cause the BIU to read the corresponding CP pointer out of the CP pointer file, increment it by 8, fetch the new CC, write it into the CC file and write the incremented CP pointer back into the file.

DMA Channel/BIU Interaction

The third source of BIU requests is from the DMA channels. When a DMA channel is active, the data flow consists of the movement of data between main memory and an I/O device. In order to moderate the difference in bandwidth between the XJS bus and the I/O devices, data is buffered in an area of RAM (Channel Buffers) on Mazda. Data movement between main memory and the data buffers occurs 32 bits at a time, while data movement between the I/O devices and the data buffers occurs 16 bits at a time. Each DMA channel monitors the state of it's data buffer and issue a request to the BIU when it's time to move a cache line to or from main memory. This request is made by setting a bit in the "Dreq" register.

I/O Control Flow

This section will describe the flow of control necessary for the execution of a Channel Program.

1) A device driver running on XJS builds a Channel Program (CP) in main memory. This program consists of Channel Commands (CC's). A CC may specify either a command operation or a data operation.

2) XJS requests execution of the Channel Program by writing the CP pointer (32 bit physical address) to the CP Pointer file on Mazda. This is a memory mapped write to an area of RAM (64, 32 bit entries) on Mazda. Writes to this RAM also cause a bit to be set in the "Xreq" register.

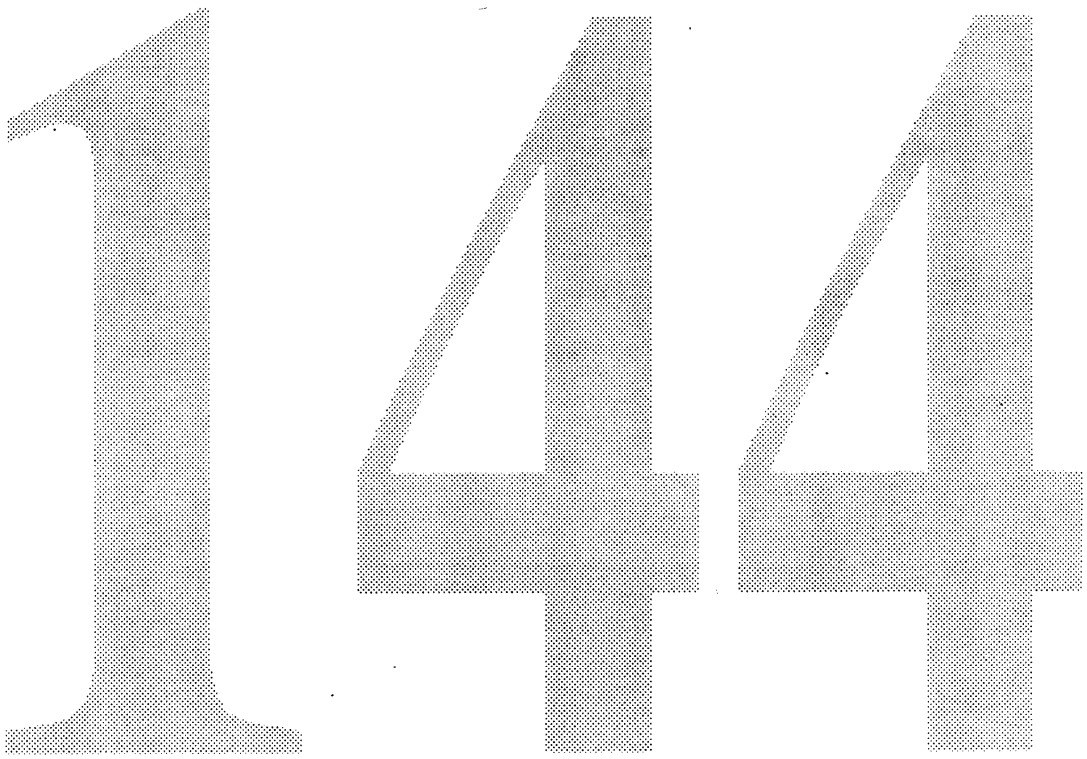
3) The Xreq register is a 64 bit register that contains one "request" bit for each of the 64 entries in the CP pointer file. A set bit in the Xreq register will cause the BIU to fetch a CC from main memory. The CC fetched is stored into the CC file. The CC file is a 64 entry by 64 bit area of RAM in Mazda. When a CC is written into one of the 64 entries of the CC file, the corresponding bit is set in the "CCrdy" register. This register is monitered by Wankel and indicates the state of each of the CC entries.

4) Wankel "tasks" are assigned entries in the CC file. When a given DMA channel is ready to accept a new command, the Wankel code for that task will check the CCrdy register to see if any bits are set for that channel. If so Wankel will read the appropriate CC, interpret the command, and do the set up for the channel. When the DMA channel is properly configured, Wankel will issue a command to activate the DMA.

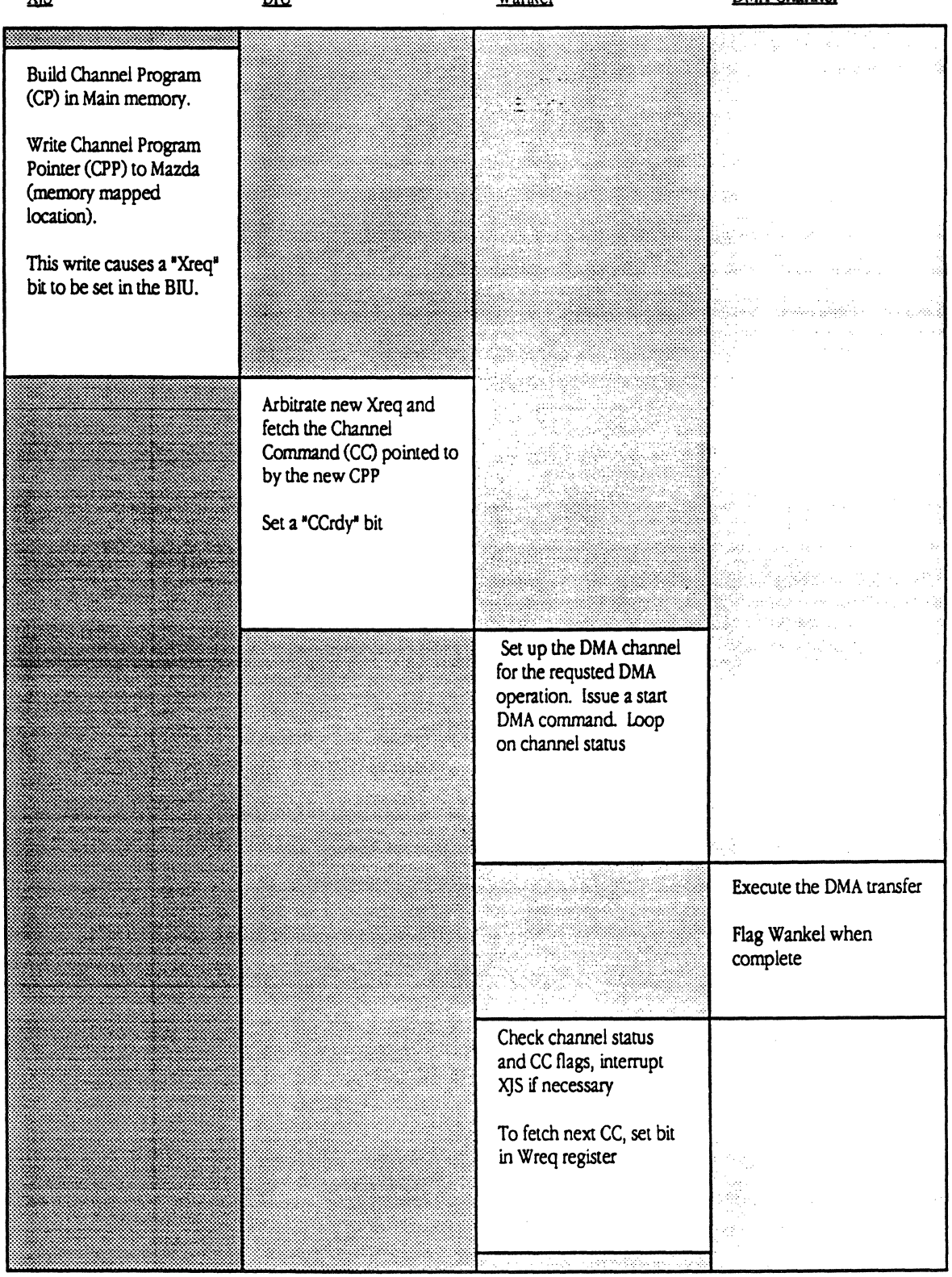
5) The DMA channel logic is responsible for the movement of data between main memory and the attached device. To achieve this it must be capable of requesting two types of service from it's data buffer (in the BIU). It must be able to move data between it's data buffer and the attached device (16 bits at a time) and it must also be able to request that the BIU move all or part of the buffer to or from main memory. Upon completion of the transfer (count goes to zero) a flag will be set to indicate to Wankel that the transfer is complete.

6) After starting the DMA transfer, the Wankel code for that channel will monitor that channel's status register. Upon seeing the completion flag, Wankel will do one of two things. If the program is complete, XJS will be interrupted. If not, the next CC will be fetched. To fetch the next CC, Wankel sets the bit in the Wreq register (64 bit register) that corresponds to the CC that just completed.

7) A set bit in the Wreq register will cause the BIU to read the corresponding address out of the CP pointer file, increment it by eight, and fetch the new CC. The incremented CP pointer is then written back into the CPP file.



Channel Program Execution



Bus Interface Unit (BIU) Operations

The following sections describe in detail how the various BIU resources must be coordinated in order to preform the required BIU operations. The resources that must be controlled include RAM structures that are shared by various functional units of the Mazda I/O controller. These RAM's include; the Channel Program (CP) pointer file (64 entries by 32 bits), the Channel Command (CC) file (64 entries by 64 bits), and the Data file (32 entries by 256 bits). The ten operations described below represent the core data movement functions required to interpret and execute Channel Programs.

CP pointer write to CPP file (in Mazda) by XJS

1.1) Bus Request (BR) set by XJS (Address Bus Busy (ABB) must be negated)

1.2) Bus Grant (BG) asserted by System Controller (SC)

1.3) XJS waits for Address Bus Busy (ABB) to be negated, then asserts ABB, negates BR , asserts Transfer Start (TS) and writes address. SC and Mazda latch address.

1.4) XJS waits for Data Bus Grant (DBG) asserted and Data Bus Busy (DBB) negated then asserts DBB and writes data

1.5) SC sends Mazda a "Slave Request Mazda" (SlvReqM) signal

1.6) Mazda writes data to CCPP file (or latches it and the required address bits in Mazda somewhere)

1.7) "Slave Acknowledge Mazda" (SlvAckM) asserted by or Mazda, received by SC (if pipelined transfers are supported, AACK may be asserted by SC as early as 1.4)

CC read from memory due to CP pointer write to CPP file (in Mazda) by XJS

2.1) BIU arbitrates internal requests

2.2) Generate address for CPP and CC file

2.3) Read CPP

2.4) BR asserted by Mazda (if ABB negated)

2.5) Bus Grant (BG) asserted by SC

2.6) Mazda waits for ABB negated then asserts ABB, negates BR, asserts TS, and writes address

2.7)* Mazda waits for DBG asserted, DBB negated then asserts DBB, and waits for TA

2.8) Mazda latches first 32 bits (Opcode, Flag, and Count) when SC asserts TA

2.9) Mazda latches second 32 bits (address) when SC asserts TA

2.10) Mazda negates DBB

*(if pipelined transfers are supported, AACK may be asserted by SC as early as 2.7)

CC read from memory due to Wankel setting a Wreq (to get the next Channel Command)

3.1) BIU arbitrates internal requests

3.2) Generate address for CPP and CC file

3.3) Read CPP and add 8 to it (and latch?)

3.4) BR asserted by Mazda (if ABB negated)

3.5) Bus Grant (BG) asserted by SC

3.6) Mazda waits for ABB negated then asserts ABB, negates BR, asserts TS, and writes address

3.7)* Mazda waits for DBG asserted, DBB negated then asserts DBB and waits for TA

3.8) Mazda latches first 32 bits (Opcode, Flag, and Count) when SC asserts TA

3.9) Mazda latches second 32 bits (address) when SC asserts TA

3.10) Mazda negates DBB

*(if pipelined transfers are supported, AACK may be asserted by SC as early as 3.7)

(somewhere between 3.3 and 3.10, the incremented CPP must be written back into the CPP file)

CC write to memory due to Wankel setting a Wreq (to write back completed Channel Command)

4.1) BIU arbitrates internal requests

4.2) Generate address for CPP and CC file

4.3) Read CPP and CC (and latch?)

4.4) BR asserted by Mazda (if ABB negated)

4.5) Bus Grant (BG) asserted by SC

4.6) Mazda waits for ABB negated then asserts ABB, negates BR, asserts TS, and writes address

4.7)* Mazda waits for DBG asserted, DBB negated then asserts DBB, writes first 32 bits (opcode and flags) , and waits for TA

4.8) SC asserts TA

4.9) Mazda writes second 32 bits (address)

4.10) SC asserts TA

4.11) Mazda negates DBB

*(if pipelined transfers are supported, AACK may be asserted by SC as early as 4.7)

Wankel read of a CC

5.1) Wankel reads the "CCreq" register (CCr/w, Reserved, CCaddr(5:0). If CCbsy is asserted, wait here. This bit is tested by reading the CCreq register and conditionally branching on the "Wankel Flag".

5.2) Wankel stores the address of the CC (6 bits) and sets the CCr/w bit by writing to the CCreq register. A write to this register also causes CCbsy to be set, this constitutes a request to the BIU

5.3) BIU arbitrates internal requests

5.4) Generate address for CC file

5.5) BIU reads CC, latches in the CC register, and sets the CCack flag in the CCdata register

5.6) Wankel reads the CCdata register (actually one of eight) if the CCack flag is set (also tested using the "Wankel Flag" then the CC is read a byte at a time. A read of any part of the CC when the "ack" flag is set will clear the CCbsy and CCack flags.

Wankel write of a CC

6.1) Wankel reads the CCreq register, if it is busy , wait here.

6.2) Wankel writes the CC a byte at a time to the CCdata register (64 bit)

6.3) Wankel stores the address of the CC (6 bits) and sets the CCr/w bit by writing to the CCreq register. A write to this register also causes CCbsy to be set, this constitutes a request to the BIU

6.4) BIU arbitrates internal requests

6.5) Generate address for CC file

6.6) BIU writes CC into CC file, and clears the CCbsy flag

DMA read from memory

7.1) BIU arbitrates internal requests

7.2) Generate address for Data file and CC file

7.3) Read CC address

7.4) BR asserted by Mazda (if ABB negated)

7.5) Bus Grant (BG) asserted by SC

7.6) Mazda waits for ABB negated then asserts ABB, negates BR, asserts TS, and writes address

7.7) * Mazda waits for DBG asserted and DBB negated then asserts DBB and waits for TA

7.8) Mazda latches first 1-4 bytes of data when SC asserts TA

7.9) Mazda continues latching data (when SC asserts TA) until transfer is complete

7.10) Mazda writes to Data file and negates DBB

*(if pipelined transfers are supported, AACK may be asserted by SC as early as 7.7)

(somewhere between 7.3 and 7.10, the updated CC must be written back into the CC file)

DMA write to memory

8.1) BIU arbitrates internal requests

8.2) Generate address for Data file and CC file

8.3) Read Data and CC (and latch?)

8.4) BR asserted by Mazda (if ABB negated)

8.5) Bus Grant (BG) asserted by SC

8.6) Mazda waits for ABB negated then asserts ABB, negates BR, asserts TS, and writes address

8.7) * Mazda waits for DBG asserted and DBB negated then asserts DBB, writes first 1-4 bytes of data , and waits for TA

8.8) SC asserts TA

8.9) Mazda continues data transfers as in 8.7 and 8.8 until transfer is complete

8.10) Mazda negates DBB

*(if pipelined transfers are supported, AACK may be asserted by SC as early as 8.7)

(somewhere between 8.3 and 8.10, the updated CC must be written back into the CC file)

DMA read from a device

9.1) BIU arbitrates internal requests

9.2) Generate address for Data file

9.3) Read Data from device buffer and write to Data file

9.4) Decrement count register and increment address register (channel registers)

DMA write to a device

9.1) BIU arbitrates internal requests

9.2) Generate address for Data file

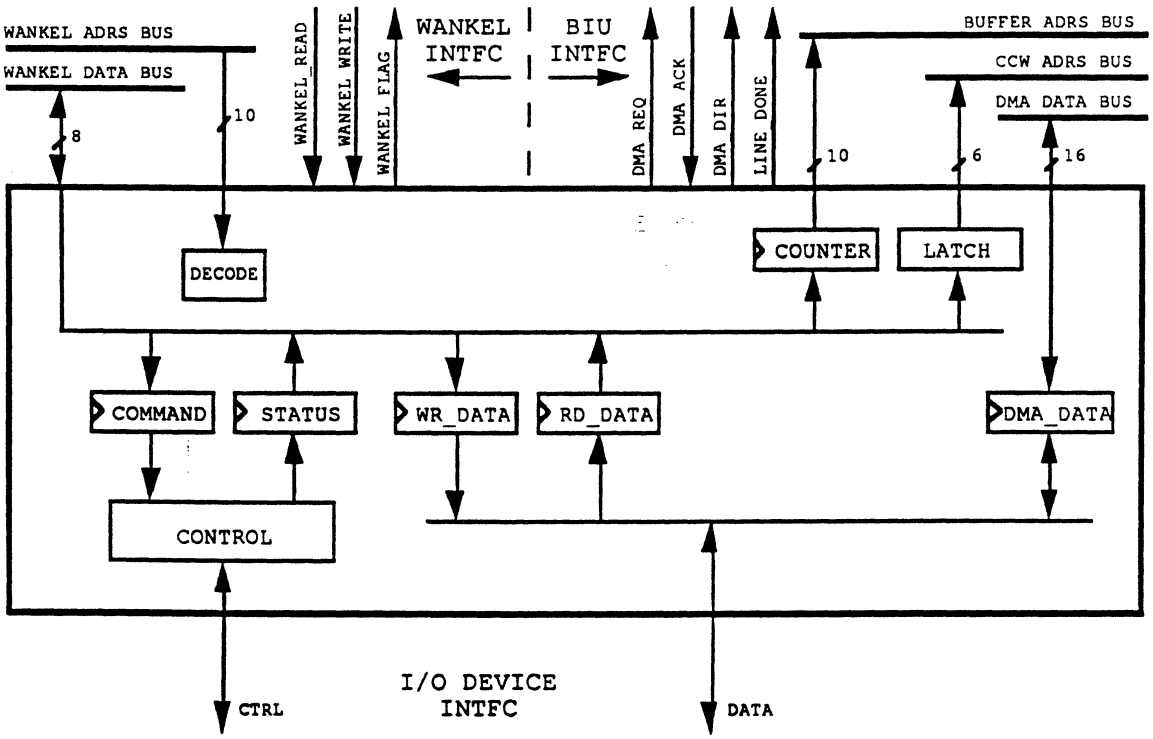
9.3) Read Data from Data file and write to device buffer

9.4) Decrement count register and increment address register (channel registers)

Generic I/O Modules

The I/O Modules on Mazda contain the individual hardware logic for each specific device interface. Each I/O Module is unique in the way it communicates with its device interface, but all of them present a similar interface to Wankel and the Bus Interface Unit (BIU). Thus it is possible to describe the characteristics of a generic I/O Module without worrying about the particulars of a specific device interface. That is the purpose of this section. For details about a particular I/O Module, refer to the section of the document dealing with the specifice device interface.

The following figure is a block diagram of a generic I/O Module. The discussion that follows is partitioned into the three interfaces of an I/O Module: Wankel, the BIU, and the I/O device.



Wankel Interface

Wankel communicates with the I/O Modules through memory-mapped load and store instructions. The read and write operations associated with these instructions are conducted over the Wankel Data Bus and the Wankel Address Bus. Each I/O Module is assigned a number of Wankel addresses for its direct-access registers (refer to the earlier section on "Wankel Interfaces" for a discussion of direct-access registers and indirect-access registers). In the generic I/O Module shown above, there are 6 direct-access registers: the Command Register, the Status Register, the Wankel Write Date Register, the Wankel Read Data Register, the Buffer Address Counter, and the CC Address Register. The typical usage of these registers is described below. Note that certain I/O Modules may require slight differences in the complement of direct-access registers.

Command Register: When a task wants to issue a command to an I/O Module, it writes this register. The decoding of the value written to the Command Register is task and I/O Module dependent. Typically the register would be split into a Command field and an optional Address field. The Address field could be used to select an indirect-access register as source or destination of any data transfer.

Status Register: Read by a task to determine status of the I/O Module.

Wankel Write Data Register: When a task wants to write an indirect-access register, the write data is written to this register first, then the Command Register is loaded with the write command and the address of the indirect-access register.

Wankel Read Data Register: When a task issues a read of an indirect-access register, the data is placed in this register when it is available.

Buffer Address Counter: When a task initiates a DMA transfer, this counter is loaded with the starting address within the BIU's DMA buffer. The I/O Module keeps this count updated as it supplies data to its DMA buffer. The count is used by the BIU to select the area of the buffer to read or write.

CC Address Register: This register is loaded by a task when it initiates a DMA transfer. The value loaded here is the address within the CC File used for the active CC. The I/O Module supplies this address to the BIU with each request to read or write memory. The BIU uses the CC Address to fetch the appropriate CC from the file for address incrementation and length decrementation.

The interface between Wankel and an I/O Module consists of the following signals. All of these signals are bussed connections between Wankel and the I/O Modules.

Wankel Data Bus: 8 bits, used for Wankel load/store data.

Wankel Address Bus: 10 bits, decoded by the I/O Module to determine whether it is the target for the load/store and if so, which direct-access register is the source/destination.

Wankel Read: set during the execution of a load instruction on Wankel.

Wankel Write: set during the execution of a store instruction on Wankel.

Wankel Flag: returned by the I/O Module during a read operation; loaded directly into the Wankel Flag status bit for immediate testing by a branch instruction.

BIU Interface

The I/O Modules arbitrate for access to the BIU over the DMA Bus. The interface between an I/O Module and the BIU consists of the following signals.

DMA Req: Each I/O Module has its own request line to the BIU. The method used to arbitrate DMA requests is described in the Bus Interface Unit section.

DMA Ack: Each I/O Module receives its own acknowledge line from the BIU.

The remainder of the interface consists of bussed signals that are driven by the I/O Module that is granted an acknowledge by the BIU.

DMA Dir: Indicates the direction of the DMA transfer

Line Done: When set, indicates that the current request completes a cache line of data. The BIU uses this signal to determine if it is to fetch/dump the I/O Module's buffer area.

Buffer Address Bus: 10-bit address within the BIU's DMA Buffer Memory. The I/O Module is responsible for maintaining this pointer and passing it to the BIU with each request.

CC Address Bus: 6-bit address within the BIU's CC File. See the description of the CC Address Register in the previous section.

DMA Data Bus: 16-bit bus used for the transfer of DMA data.

I/O Device Interface

The interface between an I/O Module and its I/O device does not lend itself very well to a generic description. All that can be shown on the block diagram is that there is a Control interface and a Data interface. The Control interface may consist of several address lines or it may consist of individual chip enables. There should be a signal that indicates direction (read/write). The Data interface may be anything from a serial line to a 16-bit wide bus.

Section 4

Mass Storage

INTRODUCTION.

Jaguar mass storage must solve traditional problems, and also new ones which arise from the new data types which will be introduced in the Jaguar environment. Among the conventional roles for mass storage, those of non-volatile data retention and protection, data distribution, and random access data retrieval and management are the most prominent.

In addition, Jaguar mass storage must function in roles which more closely resemble those of analog audio and video recording devices: the mass storage subsystems must serve as the equivalents of a VCR or tape recorder, which also happens to be random access (and digital).

This implies that the mass storage hardware and software must support continuous, sequential delivery of one or more real time streams for arbitrary intervals, and also that Jaguar mass storage is designed to minimize access time when real time data are stored or retrieved randomly, or are accessed interactively. The Jaguar mass storage system also supports the operating system's virtual memory mechanisms.

The mass storage system is organized around three principles:

- mass storage will receive and deliver real time data
- every system has a Winchester disk drive
- real time virtual memory storage requires specialized optimizations

The design supports these principles by providing low cost high performance storage, and by exploiting the flexibility of the programmable i/o controllers.

Two further points are implicit in these principles:

1) preserving real time objects' integrity requires explicit management of mechanical and command latencies; this implies closer control over the incidence of mechanical latencies in mass storage i/o than has traditionally been employed;

2) real time applications will require that a storage hierarchy covering a range of latencies and capacities be balanced by astute use of the virtual memory storage. In other words, the virtual memory storage is a means of balancing expensive high speed storage and inexpensive high capacity storage in real time situations, just as it is in traditional virtual memory systems.

Mass storage facilities are divided between those which are built into the system, and those which may (optionally) be added to the system. The interfaces and operating characteristics of these two classes of storage differ, and these differences are elaborated in the following sections.

CONCEPTS AND FACILITIES.

Architectural Components and their Attributes

The basic mass storage subsystem comprises four units: two disk drives and two controllers. The disk drives are a 3.5" floppy disk unit and a 3.5" Winchester disk drive. Every Jaguar system contains at least one of each of these drives.

The first of the two controllers is an intelligent low level formatter/sequencer which directly processes all seeks, reads, and writes to and from the floppy disk drive and the Winchester disk drive. It does not support external or optionally attached mass storage devices.

The second controller is a high level SCSI compatible protocol engine for external auxiliary storage and input/output devices.

The intelligent low level controller integrates all of the functions of a conventional disk controller which reside between the host interface and the disk's read/write channel, including error detection and correction. One controller manages up to three drives (floppy or Winchester). The controller does not implement servo functions.The intelligent low level controller permits simplification and cost reduction of the electronics resident on the Winchester disk drive assembly. The facilities of Mazda act in conjunction with the host CPU and the low level controller to implement higher level data stream management and optimization functions.

At the same time, increased flexibility in low level physical management of the drive format and geometry permits the operating system to stipulate and benefit from optimizations, especially those directed at reducing mechanical latencies, which high level controllers do not typically support. This means that the software controlling critical real time processes which involve mass storage can take measures to guarantee that disk drive latencies do not unexpectedly cause disruptions in delivery of audio and video data, for instance. Unnecessary seeking and rotation, which occur routinely in conventional low cost mass storage subsystems, can degrade quality in real time processes. A common solution to this problem, allocation of large buffers, becomes cost ineffective when high throughputs are being sustained. It is more efficient to manage the mechanical latencies explicitly, although this requires new approaches to the design of the mass storage control functions.

The SCSI control portion of the mass storage subsystem exploits the flexibility and speed of Mazda to achieve the maximum throughput contemporary high performance SCSI devices are capable of; some synchronous SCSI targets can transfer bursts at rates up to 6 megabytes per second. The SCSI controller manages the bus protocol, while Mazda governs the movement of commands and data (as formulated by the host CPU) between the host and the SCSI bus.

The SCSI interface is designed to support targets implementing either SCSI-1 or SCSI-2; SCSI-1 is a subset of SCSI-2. In addition the interface will support software and devices which implement multiple initiator environments.

Virtual Memory System Storage

The embedded Winchester disk drive is used by the virtual memory system for swapping and demand paging (no *a priori* assumptions about the nature of the operating system's implementation of these functions are made here.) The operating system may also make use of specially mapped regions of the disk (possibly formatted for the purpose) as a transient repository for real time data participating in active threads with higher bandwidth requirements than the originating medium (a network, or magneto-optical storage, for instance) could support.

The embedded Winchester's controller provides hardware support for specialized formatting of the storage regions allocated to the virtual memory system. Specialized formats permit the virtual memory system's region or regions of the Winchester disk drive to be optimized for high transfer rate and/or low latency.

In particular, zoned recording allows the number of bytes per track, and correspondingly the number of bytes transferred per second, to be higher on the outer circumference of the drive. The most time critical virtual memory storage region may therefore be located in the outermost zone.

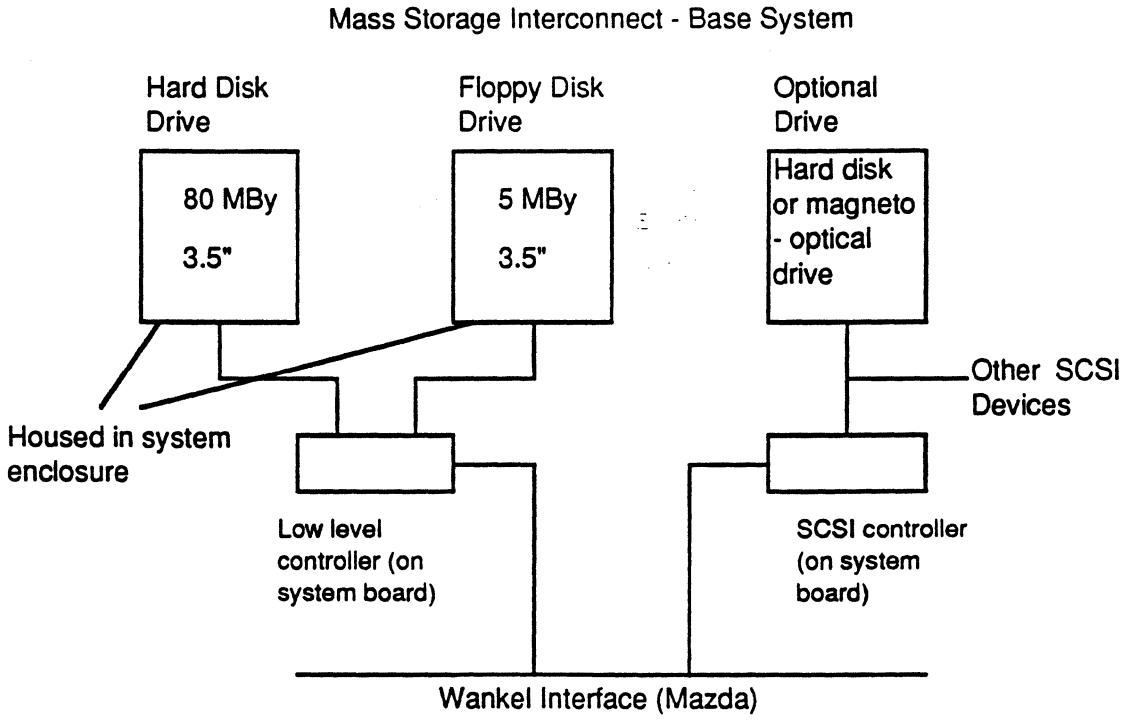
The storage hardware does not arbitrarily limit other format-dependent attributes which the operating system may adjust in order to improve virtual memory system performance. These include physical block size, and the distribution of physical ID fields, which can be tuned to optimize paging dynamics.

In addition the operating system may maintain conventional file system regions on the embedded Winchester disk drive, which may optionally be mapped into single logical entities extending to other (physical) drives.

The operating system may freely allow efficiency considerations to dictate adaptive reallocation and redistribution of file system structures and virtual storage regions between the embedded drive and additional storage facilities added to the system. In general no single physical format structure can be optimal for all possible storage hierarchies and applications requirements; the mass storage system is intended to be as flexible as possible in supporting optimization by the operating system.

Hardware Component Descriptions

The system enclosure houses one 80 megabyte Winchester disk drive and one 5 megabyte floppy disk drive. The Winchester drive is referred to as "embedded" because it is a non-optional and non-removable feature of every Jaguar system. The low level controller handles i/o to and from the floppy and the Winchester. Both the low level controller and the SCSI controller are on the main system board.



Low level intelligent disk drive controller:

The low level controller is organized around a CPU core and sets of registers through which the core CPU communicates with the host (in this instance, Mazda) and with the attached peripherals (in this instance, the floppy disk drive and the embedded Winchester drive). The registers comprising the host logical interface are described next.

Host Interface Registers

Host Data Register: 8 bits, read/write

Accesses controller data buffers and writable control store.

If host status register BUSY (see below) = 0, access to micro-controller RAM is enabled.

If BUSY = 1 and host status register SBA = 1, access to controller data buffers is enabled (controlled by internal CPU)

Block Size Register: 8 bits, write only

Allows host to select size in words of controller's dual data buffers, when host status register BUSY = 0.

Range is 1 to 256 words; block size register=words/block-1.

Size selected sets size for both buffers.

Control Strobe / Buffer Done: write only

A write by host (only when BUSY = SBA = 1) signifies finish of data buffer accesses (in cases when all locations have not been accessed).

Generates strobe, clearing SBA in host status register. (SBA is cleared automatically when all locations in buffer have been accessed.)

Mode Register: 8 bit, read/write

bit 0: enable System Buffer Available (SBA) interrupt. If set, an interrupt is generated whenever host status register SBA is set.

Clear Interrupt: 8 bit, write only

Clears pending interrupt latches. Cleared when either a) the low level controller is reset or b) the host command register is written to with EXEC set. Writing to the register causes the interrupt latch whose corresponding bit (in the register) is set to 1 to be cleared.

Bits 0-5 are reserved.

bit 6: Clear EOI

bit 7: Clear SBA

Host Command Register: 8 bit, write only

Starts execution by internal CPU when written to by host. May only be written to when SBA is 0. Bits 1-4 select internal CPU code pages 0-3 respectively. Bit 0 (EXEC) when set causes execution to commence at the first location of the selected page. This register also selects the code page to be written to when Mazda transfers new code (with EXEC = 0).

Host Status Register: 8 bit, read only

bit 0 (C/D): when clear, internal CPU expects data; when set, internal CPU expects command parameters.

bit 1 (I/O): when clear, host is to read buffer; when set, host is to write to buffer.

(bit 2 is reserved)

bit 3 (INTERRUPT pending): set when low level controller interrupt output pin is asserted. Flag is cleared when interrupt source has been cleared via the Clear Interrupt register.

bit 4 (SWAP): When set, host response to SBA should be to read the data buffer, otherwise host should write the data buffer. The bit is updated before or at the same time as SBA.

bit 5 (ERROR): set when host error register is non-zero.

bit 6 (BUSY): set when internal CPU is executing. When BUSY is set, host data register is connected to the data buffer; otherwise it is connected to the control store RAM.

bit 7 (SBA): System Buffer Available when set indicates that the buffer is available to the host for accesses. Bit is cleared when last buffer address is accessed. SBA may be used to set an interrupt latch (depends on mode register setting).

Host Error Register: 8 bit, read only

Contains code describing last fatal error during execution; valid only after BUSY goes to 0.

Host Count Register: 8 bit, read only

Any data written to this register by the internal CPU may be read by the host; it may therefore be used to report number of blocks remaining to be transferred. Contents stable only when internal CPU is not BUSY.

Peripheral Interface Registers: 16 command, 16 status registers

Four lines (SEL, and CA(2-0)) select a register; a command is written to a peripheral via the selected register by issuing a single LSTRB pulse. Status registers may be consulted while LSTRB is inactive by selecting the peripheral and asserting the appropriate address. Status is returned as a logic level on the Read Data line of the peripheral interface.

Peripheral Command Registers:

Command 0: resynchronizes serial communications between low level controller and peripheral; BUSY status at register 6 in the low level controller is set to 0.

Command 1: Strobes the bit present on ±WRDATA into the peripheral receive register. Also strobes the echoed complement on ±RDDATA into the controller receive register.

Command 2: Issues 'Seek Next Track' command to peripheral, during multitrack operations.

Command 3: Strobes bit present on ±RDDATA from peripheral into controller's serial receive register.

Command 4: reserved.

Command 5: Reset peripheral device.

Command 6: Notifies peripheral device that a serial byte is present in its serial receive register.

Command 7: reserved.

Peripheral Status Registers (all are invoked while LSTRB is inactive):

Status Register 0: read channel data are multiplexed onto the ±RDDATA line when register address 0 is asserted while LSTRB is held inactive. When -WRTGATE is asserted, the peripheral takes data to be written from the ±WRDATA line.

Status Register 1: serial communication integrity check; complement of ±WRDATA is multiplexed onto RDDATA line.

Status Register 2: 'seek complete for write' peripheral status register is multiplexed onto ±RDDATA line.

Status Register 3: peripheral's serial transmit register is multiplexed onto ±RDDATA line.

Status Register 4: peripheral's 'serial read error' status register is multiplexed onto ±RDDATA line. (Controller will then attempt a retry on the failed serial byte read operation).

Status Register 5: sense presence of Winchester by detecting low level on ±RDDATA line.

Status Register 6: read peripheral's BUSY status register; state is multiplexed onto ±RDDATA line. Commands issued by the

low level controller during BUSY state will be ignored.

Status Register 7: signals low power mode status of peripheral.

Shutdown mode means peripheral draws lowest power which will still permit it to respond to controller commands. Standby means minmum power permissible while spindle still spins, to allow rapid return to Ready status. Drive cannot seek in Standby mode.

The low level controller interface to the host (i.e., Mazda) consists of 22 pins as follows:

A(1-3): Host address bus; select a register in controller (A0 not used)

R/-W: Host read/write; determine data transfer direction between host and controller

D(0-7): Host data bus; data transfer between host and controller

-CS: Chip select; asserted to select controller for programmed i/o access by host

-DS: Device strobe; strobe data during programmed i/o access by host

-RESET: Host reset; assert to abort current operation and enter idle state

-IRQ: Host level sensitive interrupt; asserted by controller to interrupt host

-DRQ: DMA request; asserted by controller to request DMA service

-DACK: DMA acknowledge, asserted by DMA controller to select controller data register and perform DMA transfer

-DTACK: Programmed i/o device acknowledge; asserted by controller to signal that it is ready to terminate a programmed i/o bus cycle

-DREADY: DMA device ready; asserted by controller during block mode DMA when it is ready to transfer data

FCLK: Clock input; CRYS: Crystal output, if used; controller clock may be provided at FCLK from an external oscillator; or it may be provided by a crystal attached between FCLK and CRYS.

This interface may be used to transfer microcontroller instruction pages and controller parameters to the writable control store. The interface transmits command and status information and data between the host and the disk drives. No external writable control store RAM is used.

Two internal data buffers in the controller are used during data transfers between host and drives. The host has exclusive access to one of these, and the internal controller CPU has exclusive access to the other, at any one time. Each buffer has separate internal address and data paths. The internal CPU can swap buffer access. Host and internal CPU have independent and simultaneous access to their respective current buffers.

Transfers may occur between the controller and one drive at a time. Overlapped seeks may be managed within the controller by polling for seek completion. Any pending operation may be aborted by resetting the internal CPU.

The controller communicates with drives in either of two modes. It may issue commands to or obtain status information from attached peripherals via serial bytes. The same lines are used for reading data bytes from and writing data bytes to the peripherals.

The following 18 lines comprise the peripheral interface:

- ±RDDATA0, 1, 2: Serial data from peripheral
- -WRGATE: Request to write
- ±WRDATA0, 1, 2: Serial data to peripheral
- -ENBL0, 1, 2: Enable peripheral
- SEL: Floppy head select
- CA(0-2): Peripheral control address
- LSTRB: Perform peripheral command
- -SERVO: Servo field (or general purpose)
- -INDEX: Index pulse from Winchester
- -ATTN: Error/status from peripheral

The Winchester disk drive supports commands which may be invoked by the controller using the serial command interface. Commands and status are transmitted between the peripheral device and

the controller's peripheral interface registers. LSTRB and CA(0-2) select the register to be used in a transaction.

The controller cannot support concurrent transfers to two peripherals. System software must therefore ensure that no uninterruptable transfer operation of lower priority be launched when it might interfere with initiating a higher priority transaction (especially to or from the virtual memory storage device).

The host may communicate data to or from its currently selected buffer while an unrelated i/o operation to a peripheral is in progress. Multiple i/o paths may thus be active simultaneously.

Floppy Disk Drive Specifications:

The floppy disk drive included with each Jaguar system will support high capacity media (barium ferrite) which the low level controller can format to approximately five megabytes. At the highest data density the floppy format is based on a run length limited code which differs from the formats previously used on Apple machines.

In addition to the five megabyte format, the low level controller is capable of formatting, reading, and writing the lower density 3.5" formats in use on Apple computers, including MFM formats. The floppy disk drive will at minimum be able to read and write older MFM formats, and it will read the older GCR formats. Writing older GCR formats is still under investigation.

Floppy disk drive summary specifications (preliminary):

Power:

+12.0V ±5%	Standby -	40mA (motor off)
	R/W -	185mA Max
	Motor Start -	600mA Max
+5.0V ±5%	Standby -	210mA
	Typical -	210mA

MTBF: 30,000 POH

MTTR: 30 Minutes

Hard read errors: 1 per 10**12 bits read

Floppy disk drive interface signals:

CA(2-0) - Addressable latch selection

/ENBL - Enable

LSTRB - Line Strobe

RD - Read Line

/WRTGATE - Write Gate

WRTDATA - Write Data

/CSTOUT - Cassette Out

/EJECT

Floppy Disk Drive Controller Interface Signals:

GND	1	2	CA0
GND	3	4	CA1
GND	5	6	CA2
GND	7	8	LSTRB
/EJECT	9	10	/WRTGATE
+5V	11	12	SEL
+12V	13	14	/ENBL
+12V	15	16	RD
+12V	17	18	WRTDATA
+12V	19	20	/CSTOUT

Embedded Winchester Disk Drive Specifications

The embedded Winchester disk drive is semi-permanently attached (not user removable) to the system housing and board. Its presence is a precondition for normal Jaguar operation (only limited diagnostic functions will be supported in the absence of a working embedded Winchester disk drive).

The drive has a nominal formatted capacity of 80 megabytes, and conforms nominally to a 3.5" wide by 1" high form factor.

Winchester disk drive specifications (preliminary):

Power:

12V +10%/-5%, 5V ±5%

		Mean	Maximum	
Startup -		7.0W		10.0W
Random R/W -	3.0		3.5	
Ready -		1.5		2.0

Standby -	1.2	1.5
Shutdown -	0.5	0.8

Capacity:

Sector Size -	512 bytes
Formatted Capacity -	>= 160,000 sectors
Disks -	2
Heads -	4

Performance:

Average Access Time -	<= 20ms
Head Switch Time -	<= 4ms
Average Latency -	<= 9ms
Power On, until Ready -	<= 8s

Dimensions:

Length x Width x Height -	150 x 102 x 25.4mm

Weight: n/a

Reliability:

MTBF -	>= 50,000 POH
MTTR -	<= 30 min.

Winchester Disk Drive Interface to Low Level Controller:

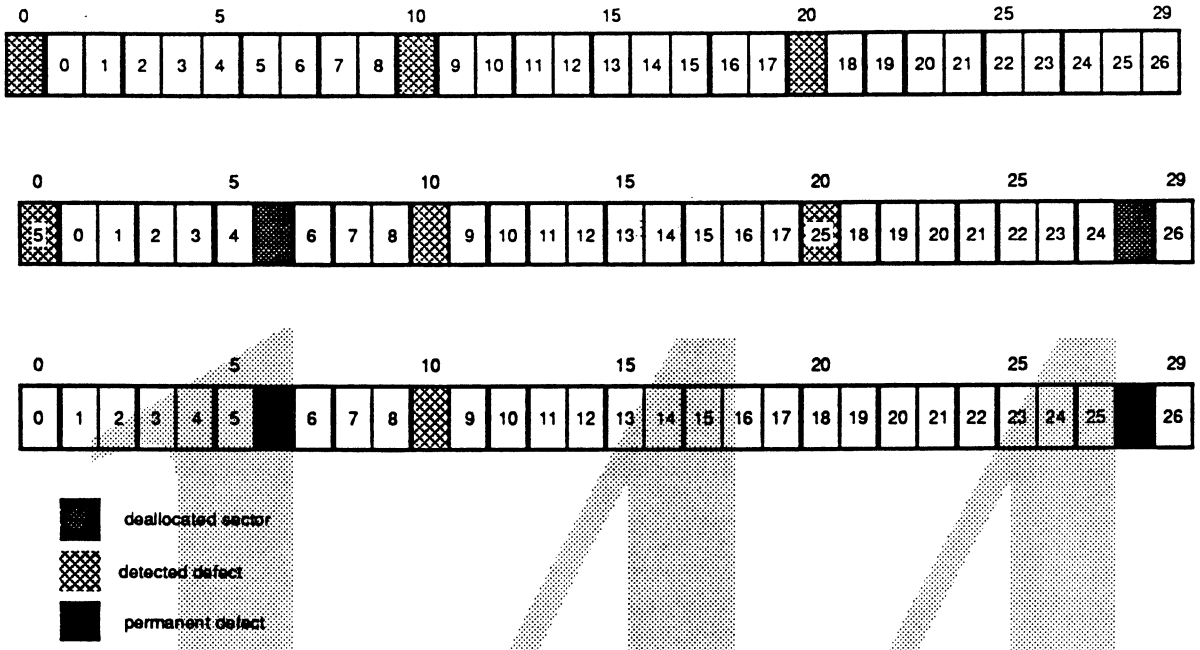
GND	1	2	CA0
+12	3	4	CA1
+12	5	6	CA2
+12	7	8	-ENBL
+12	9	10	-WRTGATE
GND	11	12	-SERVO
+5	13	14	-ATTN
+5	15	16	GND
+RDDATA	17	18	-RDDATA
GND	19	20	GND
+WRDATA	21	22	-WRDATA
GND	23	24	-INDEX
GND	25	26	LSTRB

Embedded Winchester Defect Management:

The low level controller governs the detection, correction, and remapping of hard and soft errors arising during Winchester disk i/o operations. The low level controller implements an error correction scheme based on a 48 bit polynomial. Error correction is performed by the internal CPU. When a correctable error is detected, the low level controller will initiate loading of any code pages needed for carrying out the correction algorithm.

Spare sectors are reserved on each track. If a sector is found during formatting or subsequently to contain a hard defect according to the analysis algorithm, it is remapped to one of the spare locations, and if it had been written to, its content is corrected (if possible) and written to the new location.

Eventually, or immediately if the defect lies in a time critical data path, the track may be further adjusted by reordering the user sector sequence to correspond to the logical sector sequence on the track. This could eliminate multiple superfluous rotations in the worst case. Sectors containing hard defects are remapped to "defective" status and are skipped over during i/o operations; they no longer possess logical addresses and are disregarded.



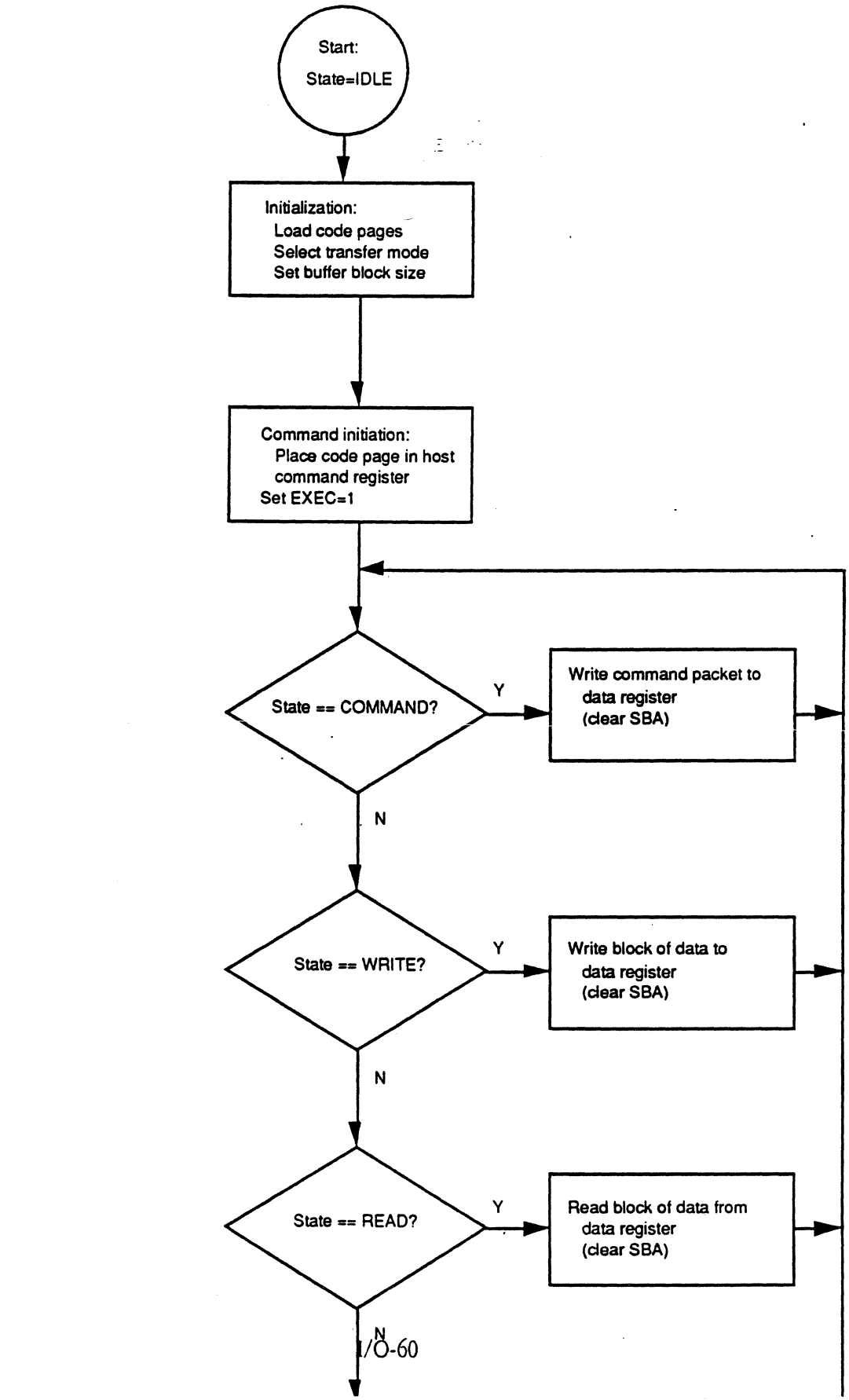
DATA TRANSFER CONTROL FLOW EXAMPLES

Example Data Transfer Using Low Level Controller:

(Initialization: the controller's writable control store is initialized by the host as follows. The host detects that the internal CPU is not busy, provides a page address in the host command register which also sets the internal program counter to zero, and writes the instruction for the first location in that page to the host data buffer.

The writable control store is not accessible through the DMA facility. The internal program counter is automatically post incremented. The high two bits of the instruction word are stripped away. Loading continues until the page is exhausted. Segments may not be spanned in a single transfer sequence. Programs may not span segment boundaries.)

Low level controller command execution:



Having initialized the internal writable control store with the page or pages of code necessary for carrying out the intended operations, the host initiates a command in the low level controller. When SBA in the host status register is 0 and the internal CPU is not BUSY, the host writes to the command register, setting bit 0 (EXEC) and the bit (4 - 1) selecting the code page for the operation to be performed. Execution commences from location 0 in the selected page. Setting EXEC also clears the error register and interrupt latches, and causes busy to be set in the status register.

The internal CPU is wholly responsible for controlling the movement of commands, status and data between the selected peripheral and itself. Once data have been placed in the system buffer (assuming a read operation), the host is notified by the internal CPU via the SWAP, SBA, and INTERRUPT bits in the host status register that it may begin transferring the requested data.

Specifically, SWAP is set to indicate that the host should read the system buffer; this is done at the same time SBA (System Buffer Available) is set. SBA also sets an interrupt latch.

If the pending process must be pre-empted by a critical process of higher priority, the host asserts reset by writing any data to the Control Strobe register (port). This has the effect of immediately terminating execution and clearing interrupt latches, host command register, host mode register, host error register, and all bits of the host status register except SWAP (i.e. host status register is set to 0x10).

To take the example of a seek command, the host causes execution of the appropriate page to begin. The internal CPU selects the drive and issues the code for a seek command (0x0c) and tests status register 6 (BUSY) with LSTRB inactive, for completion of the command.

If a second drive is also to initiate a concurrent seek, the host may transfer the logical address for the second seek command and cause the internal CPU to execute another seek. Selecting another device automatically causes deselection of the first unit, which is still seeking.

The internal CPU polls status on the drives (by successively reading status register 6 until BUSY is deasserted) and interrupts the host by asserting SBA when a drive has completed its seek. A 'seek complete' protocol status is reported to the host.

The host has the option of preloading a buffer in the controller while a seek is underway if one of the drives seeking will ultimately perform a write operation. (Internal CPU actions and host operations on the buffer are wholly independent of one another.) If a read is known to be required first, there is no benefit to preloading a buffer, since both buffers are required during reads in order to maintain one to one interleave performance.

Once a seek has been successfully completed, subsequent seeks may be issued relative to the current location as single instructions by the internal CPU.

SCSI Interface:

The Small Computer System Interface (SCSI) connection is via a protocol engine attached to and under the control of Mazda. The engine implements the low level protocol management features of the ANSI X3T9.2 draft proposal of 16 March 1989. The data bus interface to the host (i.e. to Mazda) is 8 bits wide.

The following general characteristics of the SCSI protocol and draft specification are supported (some of these imply one another):

- multiple initiator operation
- arbitration
- disconnect and reselection
- command queuing
- linked commands
- synchronous block data transfer
- bus parity

No feature of this implementation should prevent attaching an otherwise unsupported target device which correctly implements the SCSI-1 protocol, and for which a suitable driver is provided.

SCSI Signals

SCSI signals (on the SCSI bus side) are actively driven true (asserted). Non-OR-tied drivers may actively be driven false (negated).

Host (Mazda) Interface to SCSI Controller:

- A(3-0)
- -CS
- -IOR
- -IOW
- -IRQ
- DRQ
- -DACK
- -RST
- DB(7-0)

SCSI Bus (Peripheral Interface):

- BSY (Busy) -- OR-tied only
- SEL (Select) -- OR-tied only

- C/D (Control/Data)
- I/O (Input/Output)
- MSG (Message)
- REQ (Request)
- ACK (Acknowledge)
- ATN (Attention)
- RST (Reset) – OR-tied only
- DB(7-0,P) (Data Bus and Parity)

SCSI Read Command Example:

1) Operating system allocates memory for data to be read, and for status information to be passed from target.
2) Driver creates or initializes a command descriptor block in static memory defining the operation, including the op code.
3) The initiator (host) transfers pointers to the command block, data and status areas, and the SCSI device address, to Mazda.
4) The SCSI controller is directed by Mazda to arbitrate for the SCSI bus.
5) When arbitration is won, the device whose ID was passed is selected by the SCSI controller with ATN asserted (signifying that MSG phase follows).
6) Upon responding to the selection phase, the target reacts to the assertion of ATN by receiving the message IDENTIFY from the initiator.
7) The logical unit number is passed to the target by the IDENTIFY message, and Mazda signifies its ability to support disconnect and reselect.
8) The command descriptor block to which Mazda had been passed a pointer by the host CPU is now transferred to the target by the SCSI controller.
9) The target determines that it should disconnect from the SCSI bus while it seeks to the requested track. The target sends the messages SAVE DATA POINTER and DISCONNECT to the host.
10) The current data pointer is saved by Mazda in response to the message from the target.
11) After the DISCONNECT message is sent, the target releases BSY, freeing the SCSI bus.
12) When it has begun reading the data requested by the host, the target arbitrates for the bus and reselects the host SCSI controller. It then sends an IDENTIFY message to indicate which logical unit is re-establishing connection to the host.
13) The target transfers data to the host; Mazda directs the data to the buffer to which a pointer was passed at the outset of the operation.

14) If the target detects that another seek is required in order to complete processing of the request (at a cylinder boundary, for example), it will repeat the disconnect and reselect sequences, again issuing the SAVE DATA POINTER message to Mazda so that the transfer may resume at the point the disconnect was initiated.

15) When all data requested have been transferred, or when the command has terminated prematurely for some reason, status is sent to Mazda via the SCSI controller, and the target issues the message COMMAND COMPLETE, and goes to the BUS FREE phase.

16) Mazda interrupts the host with notification that the command is complete.

Methods:

This section discusses some ways the Jaguar mass storage facilities might be used by higher level system software resources. The problem of efficiently managing real time i/o streams is the main motivation for this preliminary overview.

The problem of managing real time i/o on mass storage devices has two main components: first, mass storage devices are subject to latencies which are very large in proportion to their burst data transfer capability; and second, that the sustainable serial bit rate from a single mass storage transducer (even disregarding the latency problem) is insufficient for some classes of storage i/o (especially video, but also static images and real time instrumentation, for instance).

Thus Jaguar's base mass storage architecture must offer means (which the operating system and applications need not necessarily exploit) for constructing controlled streams in which the incidence of catastrophic latencies is, where possible, anticipated, and in which aggregated storage devices may be combined seamlessly (from the standpoint of the user and of applications).

To enable the hardware to allow the operating system to fulfill critical real time requests in a timely manner, maps can be maintained containing the information necessary to formulate at any instant the optimal fulfillment path through the components of all outstanding requests. Such maps would also be of use in formulating policy for guiding graceful degradation in cases of i/o saturation. They also could play a role in making it easy for the operating system to synchronize multiple streams from independent sources.

This process entails examining the following for each prospective transfer operation, or perhaps group of operations:

- the duration of the operation if all of it were completed without interruption;
- the incidence of latencies in the path of the intended operation (seeks, rotational delays, head switching time, command overhead, transfer time, etc.);
- the sensitivity of the operation to segmentation during its fulfillment;
- the criticality of preserving the temporal integrity of the operation during some or all of its phases.
- the proportion of available system resources which the operations' phases would consume.

Traditional block oriented storage hardware and i/o processes are responsive to these issues only to the extent that zero latency full track reads, overlapped seeking, sector interleaving and cylinder staggering are supported.

The temporal relationships implied by different policies for fulfilling multiple simultaneous pending requests cannot in general be analyzed in traditional storage i/o architectures. Usually low cost SCSI drives today maintain on-board buffers capable of storing one or more tracks, so that the burst transfer capacity of the i/o bus can be used more efficiently. However this still does not allow system software to predict and manage the incidence of latencies when high bandwidth sustained streams are created to or from the device(s).

In a multi-threaded real time storage subsystem the operating system cannot in general acquire and process quickly enough all of the information necessary to optimally organize policy for fulfilling queued requests, in part because of the latencies associated with issuing the requests themselves, and in part because the physical behavior of the drive is so completely hidden (by design) from the data transfer interface with which the operating system and interface hardware interact.

The hardware and embedded code must therefore be organized in such a way as to update and maintain this information in a form the operating system can use easily and efficiently in determining how best to fulfill the pending i/o requests, some or all of which may be time critical.

In the Jaguar mass storage system, the low level controller in conjunction with the Winchester drive supporting the virtual memory system provides a very tight coupling between the formulation of i/o requests by the operating system and the physical behaviors which are occurring before and during the request's fulfillment. The drive is therefore in a position to serve as a delivery vehicle for staged data drawn from slower facilities with higher latencies (networks, magneto-optical storage), as mentioned earlier. The uses to which these mechanisms are put will depend on the design and implementation of the operating systems running on Jaguar.

In Jaguar's storage hierarchy the embedded Winchester will be subject to closer real time control than any other storage device because the low level controller gives provides a direct link to its behavior at the block level. This means that operating system facilities for regulating the progress of real time data transfers may elect to interpose the embedded Winchester in order to make use of this higher degree of control. In many real time situations it is more important that latencies be well characterized than that they merely be very small.

Section 5

Network/Telecom

Introduction

Computer communications was a technical novelty in the 70's marketplace. Apple harnessed the complexity surrounding this technology in the 80's with tremendous success. We have the momentum and expertise to shape and direct the evolution of telecommunications in the 90's. Jaguar will introduce the concept of truly *global desktop* to personal computing experience.

The only certainty is that the personal computer landscape will change dramatically in the coming decade. Jaguar is designed to anticipate those changes, and to give developers the preferred environment for evolutionary applications development.

The following trends are expected to unfold in the 90's:

- Increased mobility: An increasing amount of work will be done away from the office. This will place increased focus on "plain old" dialup phone service, whether it be from home office, work site, or cellular telephone.
- FAX usage will increase tremendously. Dialup FAX info/shopping/banking services will intensify this trend.
- Global connectivity: Computers will be able to communicate regardless of distance or country. But this won't happen until ease of use approaches of a phone call. Converging networking technologies (eg., X.400 E-MAIL, OSI, TCP/IP) and regulatory environments (eg EEC 1992 convergence, COS, CCITT) will help make this a reality.
- LAN clusters: LAN networks will continue to grow in sophistication, and will form the basis of many corporate workplaces. High speed backbone connections (T1, fiber, etc) will interconnect the LAN clusters. Routers will be play an increasingly important rôle in the interconnection of different LAN and WAN technologies. Dialup LAN access will become commonplace as a result of telecommuting and worker mobility.
- ISDN deployment: ISDN Basic-Rate connections (2B+D) will find growing use, especially in small and medium sized businesses.
- Telecom Agents:High-performance CPUs will cause voice input/output technology to mature in the 90's. Machines will be judged not by raw speed but by "personality": that is, friendliness and the potential for useful work away from the office by mobile information workers.

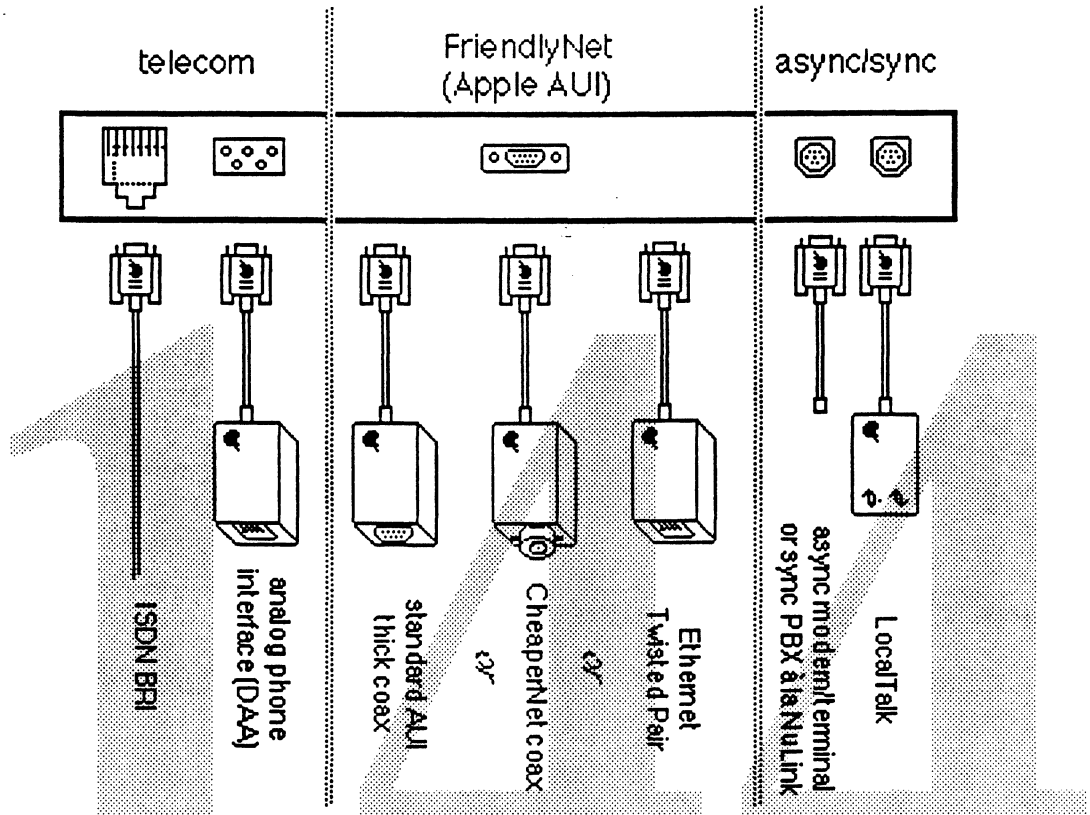
- PBX technology will compete with LAN and ISDN services. Many larger corporate customers have large financial commitments to their in-house systems; long-term migration will be away from the PBX.

Considering these trends, the Jaguar Net/Telecom goal is to allow our customers to freely exchange ideas and information, regardless of location, distance, means of communication, *or proximity to machine.*

The Network/Telecom subsystem is Jaguar's gateway to the world. It includes all communications facilities that are standard to the Jaguar; card-based communications (eg., FDDI, token ring, X.25) controllers will be adapted from N&C technology are not treated here. The major components of the Network/Telecom subsystem are:

- RALPH: the Real-time AnaLog PHone. RALPH is the analog telephone interface which comprises an analog interface system (ADC, DAC, and AGC), telephone network interface circuitry (the Data Access Arrangement or DAA), and control software for realization of modem pseudo-devices.
- BRIAN: the ISDN basic rate interface, which provides Jaguar with ISDN TE1 (Terminal Equipment 1) services on the ISDN S/T (System/Terminal) interface.
- SPAM - the Signal Processing Access Manager: a low-level DMA buffer management task that controls the flow of real-time digital sampling systems inside Jaguar.
- Apple "FriendlyNet" LAN interface which provides an Apple AUI to support the Ethernet variants 10 Mbps thick cable, 10 Mbps thin cable, and (subject to further study) "10 Base T" or 10 Mbps twisted pair.
- LocalTalk interface.
- Asynchronous communications.
- Limited synchronous communications.

The diagram below is a conceptual view of the Jaguar CPU's "business end". The following sections will describe in turn how each is implemented.

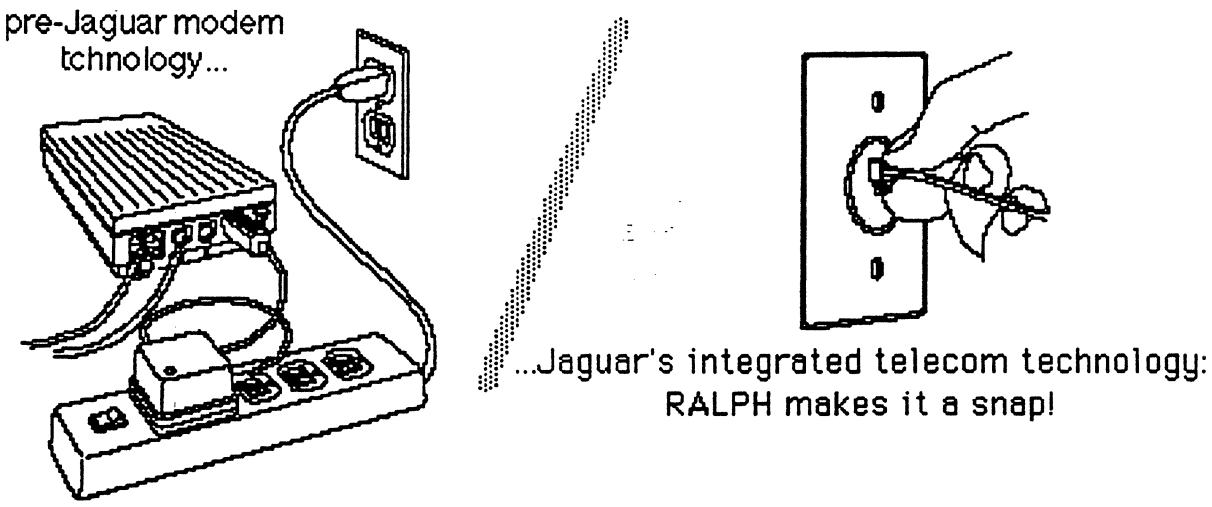


Despite Jaguar's tremendous communications potential, the goal is not to replace the telephone set. Instead, Jaguar will complement existing analog phone and PBX technology, while exploiting new digital ISDN services - thus representing a very powerful, remotely accessible idea routing tool.

RALPH: The Real-time AnaLog PHone

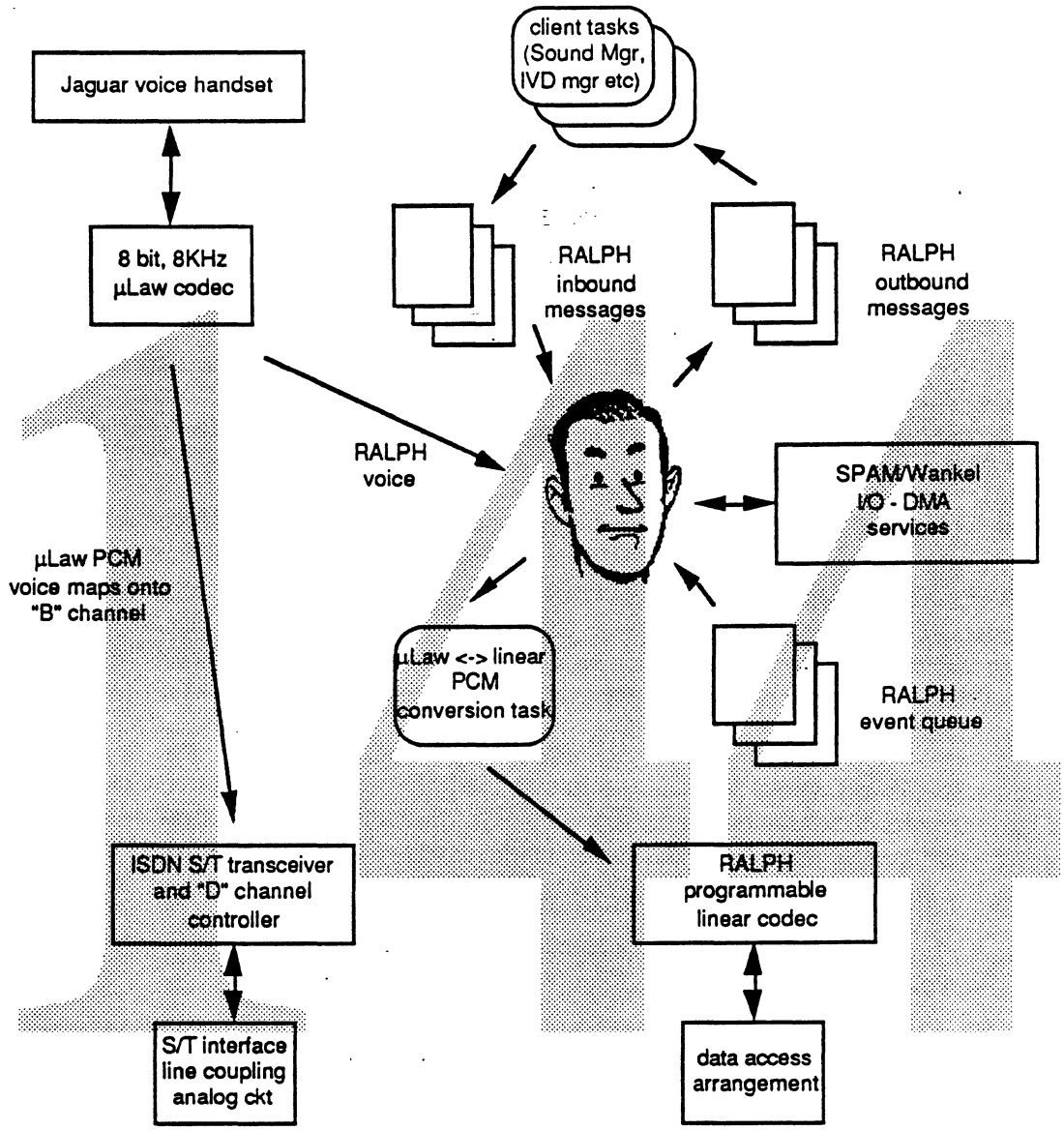
RALPH provides a high performance interconnect to the world's analog phone system. This interface is designed to support the following functions:

- Telephone-quality (300-3KHz voice grade) speech input/output
- Modem emulation under software control, including but not limited to the 300/1200/2400, v.27ter (2400/4800 bps), and v.29 FAX (7200/9600 bps) standards
- Flexible DTMF encode/decode support as a foundation for voicemail and voice/data applications.



RALPH Operational Overview

The diagram below shows the relationship of RALPH with Jaguar hardware and system software.



RALPH is a telecom-oriented, low-level application team that provides voice and data facilities to clients via its message-oriented interface. Real-time scheduling requirements are severe, as buffer underruns occuring while acting as a modem pseudo-device can cause loss of synchronization. RALPH uses SPAM - the signal processing access manager's - to route signal buffers between the Jaguar CPU and physical interface.

RALPH operates as an access manager that monitors the telecom system on behalf of client application programs. The lowest level RALPH-client interface uses asynchronous messaging IPC. Higher level C++ objects will be built on top of this foundation for direct exploitation byapplication programmers.

RALPH maintains a queue of telecom events and passes these on to its client. These events include:

- Ring Detect
- DMTF sequence received

- modem pseudo-device carrier detect/drop
- demodulated data stream

RALPH clients access RALPH services using conventions established by the pink I/O Name Server and Access Manager. These Net/Telecom Access Manager "hides" the implementation-specific details of the net/telecom system from the application programmer. For example, it should be transparent to the application if a "real" external modem were connected to the serial port and substituted for a RALPH pseudo-device modem. The parent Access Manager will enforce an exclusive access policy on RALPH pseudo-devices; data stream distribution must be performed outside the Access Manager.

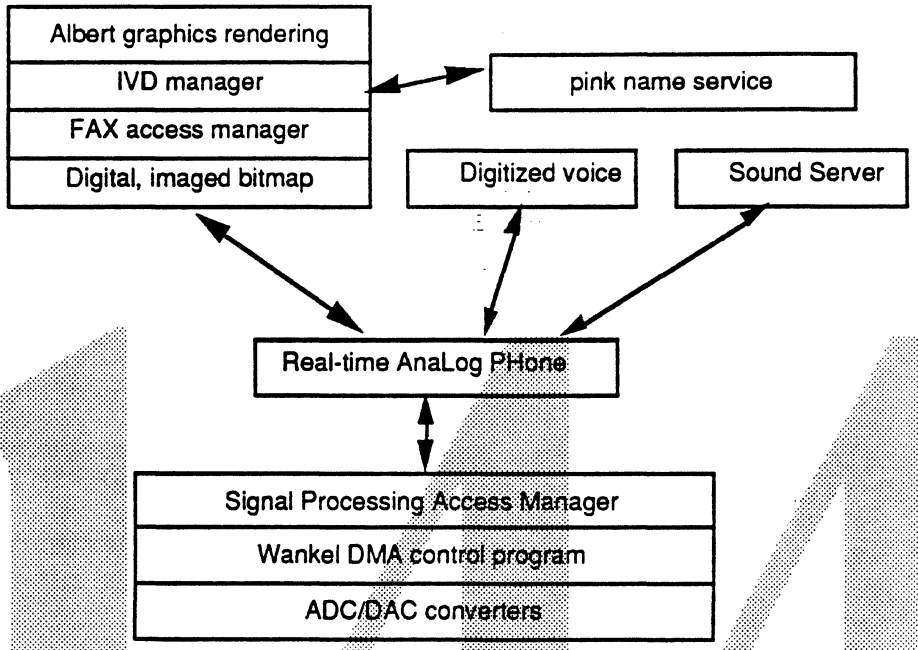
An application programmer would not access RALPH services directly. Instead, existing higher-level communications managers such as the Communications ToolBox or Integrated Voice Data Manager would become RALPH's client, thus shielding the application programmer from a (very) hardware specific implementation.

The Communications ToolBox and Integrated Voice Data Manager represent a procedural interface "shell" that attempts to hide implementation details from the application programmer. These concepts will form the basis of a Net/Telecom Access Manager. Pink has introduced the idea of an object-oriented Sound Manager and Sound Server. Among the objects currently proposed by the Sound Manager are a generic telephone line and modem. The Sound Manager objects will support voice communications, voicemail recording/playback, and the like. Sound Manager methods using RALPH and SPAM primitives will be created to support these applications.

Let's take a specific example of how RALPH would be used in a practical situation. Suppose that the user has a graphics document to send cross-country. The operation could be instigated by the user dragging a document to some iconic destination on the desktop - or by verbal request via the headset. This would result in the following events:

- the document is imaged into a bitmap by Albert, which is then presented to the Integrated Voice/Data manager.
- the IVD manager consults the pink name service to obtain the destination address. This is passed to the fax access manager.
- If the destination corresponds to an ISDN address, the bitmap will be translated into a group 4 compressed bit stream. If the address corresponds to an analog phone number, then the bitmap's translated into a group 3 compressed bit stream RALPH.
- A group 4 fax is sent over an ISDN channel via BRIAN, the basic-rate ISDN access manager. A group 3 fax is sent over an analog phone channel via RALPH, emulating a V.29 modem. spam is used to route sample buffers between the I/O system and the BRIAN or RALPH access managers.

The following diagram shows where RALPH and other components of the Network/Telecom subsystem fit into the processing hierarchy in this example:



RALPH is responsible for the following tasks:

- routing digitized voice traffic to and from the human interface and ISDN or analog phone line.
- emulation of modem pseudo-devices. RALPH encodes the input bit stream into modulated signal and decodes received modulation into a bit stream.
- PCM encoded data conversion between linear and μ-law or A-law companding (the latter being used outside the US and Japan)

Note that RALPH is *not* responsible for higher level protocol tasks, such as:

- FAX bitmap encode/decode.
- file transfer protocols
- data compression or encryption

These services will be provided by other access managers to be developed for the Network/Telecom subsystem.

The Jaguar telecom philosophy is that communications services should be interchangeable if feasible within the constraint of time. For example, whether the client sends a FAX over ISDN or analog phone line should be transparent to the user; the sole distinction being the transmission time. From the user's perspective, an external modem should operate identically to RALPH pseudo-device modem.

Let's take the example of an application that wants to transmit a fax image. This entails the following sequence of events:

- Application calls Integrated Voice-Data Manager to establish connection with remote station. Note that the IVD hides the details of the remote station's characteristics: it could be an ISDN terminal, external modem connected on the async port, or V.29 group 3 FAX
- The IVD configures RALPH by issuing the DoCommand() procedure. This translates into a message containing an ASCII-encoded string of commands that could include going offhook, DTMF tone generation, selection of modem pseudo-device to emulate.
- RALPH asynchronously returns a message to the IVD containing status reflecting the outcome of the associated request. In this case it could be "connected", "not connected", "busy", or "modem pseudo-device in use".
- application can now proceed to transmit encoded image over the modem pseudo-device.

RALPH operates in similar manner when answering the phone. In this case, however, it may not be known *a priori* whether the caller is voice or modem. There are two alternatives in this case:

- RALPH is instructed to answer the phone assuming a specific mode of operation. This would mean using a predefined modem protocol (eg. FAX) or silence in the case of anticipated voice calls).
- RALPH uses a pre-defined sequence of modem initiation sequences to try and establish communications with the (presumed) originating modem. Note that modem standards provide for "graceful" determination of calling modem type.

Neither of these two alternatives permit intelligent switching between the domain of voice and data. RALPH would lock Jaguar into a specific communications session (voice or data) without the possibility of dynamic switching. It would be much better if RALPH could exercise some intelligence in "answering the phone".

One method providing more flexibility would work as follows. On answering the phone, RALPH would play a prerecorded message that's stored on disk. RALPH would then "listen" to the line for a short period of time (say three seconds), expecting a response. The response could be voice energy (the caller's voice) or a DTMF sequence: DTMF signalling would be simplest and least ambiguous method of routing the call. At its simplest, any DTMF tone response could be interpreted as "caller wants to leave a voice message". More sophisticated users could create a more elaborate parser using the by-now familiar "tree walk" paradigm: "press 1 to ring user, press 2 to leave voicemail, press 3 to consult mr pink...". The eventual (and I believe achievable) goal is to permit relatively structured, yet extremely powerful voice I/O via Jaguar's voice daemon.

If Jaguar's message is received with silence, Jaguar knows that the caller is must be a modem in originate mode (or crank call!). In either case Jaguar will respond with a modem training sequence according to the pre-defined sequence mentioned above. This technique will work because the Jaguar voice message playback should not be "audible" to a calling modem; it's expecting the answering modem's audio handshake sequence which is a very structured signal. Further, the originating modem always remains silent until the answering modem's sequence is recognized, so even simple measure of audio energy in the voice channel would suffice for differentiating a human from a modem.

The Jaguar voice daemon will provide truly useful remote access capabilities. Imagine, one's machine can be given a recognizable "personality" through acquired recognition skills, vocabulary, speaking (output) accent, behavior...

RALPH standby mode

RALPH doesn't exist when power's been cut. What strategies could be used to answer the phone when the machine's off? As mention elsewhere in the Jaguar ERS, Jaguar has a "soft" power-off mode: user perceives the quiescent machine to be "off", but in fact MAZDA and RAM are active. This feature permits RALPH to detect a ring indication on the phone DAA and activate the machine. Here's a scenario on how this would work.

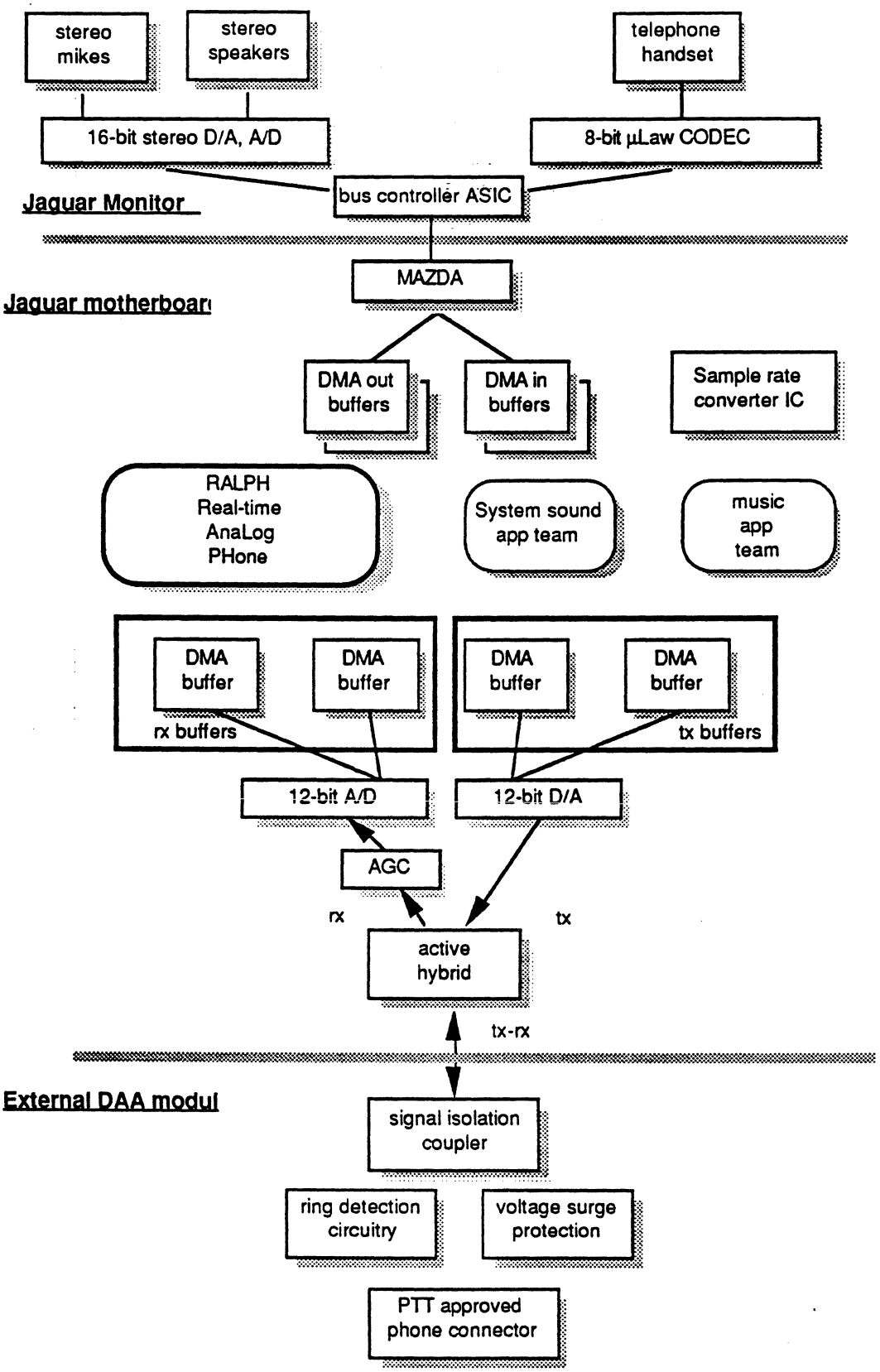
The user will use a simple "message capture" utility (using standard Jaguar hardware) to record the outgoing message. This and other messages are placed in a library on disk. The outgoing message is read from disk and stored in a specially allocated RAM area as part of the shutdown process (ie., transition to "soft" power-off mode). MAZDA is then instructed to begin execution the standby WANKEL telecom control program. This program instructs the WANKEL channel program to wait for an incoming call. When a call arrives, the WANKEL control program will "answer" the phone (by setting the DAA interface bit), activate RAM, play the answerback message through the codec to the caller (via the MAZDA DMA controller), all while concurrently "booting" the system.

The proscribed ringing sequence for dialup service is six seconds per ring (two seconds "ring" and four seconds silence)[1]. Assuming that five rings is a reasonable limit to wait for the called party to answer, Jaguar would have thirty seconds to answer the phone and begin playback of the outgoing message. A system reboot from disk within this limit appears to be feasible. More study is needed in this area.

RALPH hardware

RALPH is a telecommunications-oriented access manager that provides a programmatic interface to a universal modem "data pump". It consists of an application "team" with associated interrupt service routines that establish, maintain, and terminate voice and data connections using Jaguar's built-in hardware. A block diagram of the RALPH subsystem is shown below. The principal hardware components of this system are the data access arrangement, analog interface subsystem, and automatic gain control.

[1]the PBX standard ring sequence is four seconds (1 "ring" and 3 silence).


stereo mikes
stereo speakers
telephone handset
16-bit stereo D/A, A/D
8-bit µLaw CODEC
bus controller ASIC
Jaguar Monitor
MAZDA
Jaguar motherboar
DMA out buffers
DMA in buffers
Sample rate converter IC
RALPH Real-time AnaLog PHone
System sound app team
music app team
DMA buffer
DMA buffer
rx buffers
DMA buffer
DMA buffer
tx buffers
12-bit A/D
12-bit D/A
AGC
rx
tx
active hybrid
tx-rx
External DAA modul
signal isolation coupler
ring detection circuitry
voltage surge protection
PTT approved phone connector

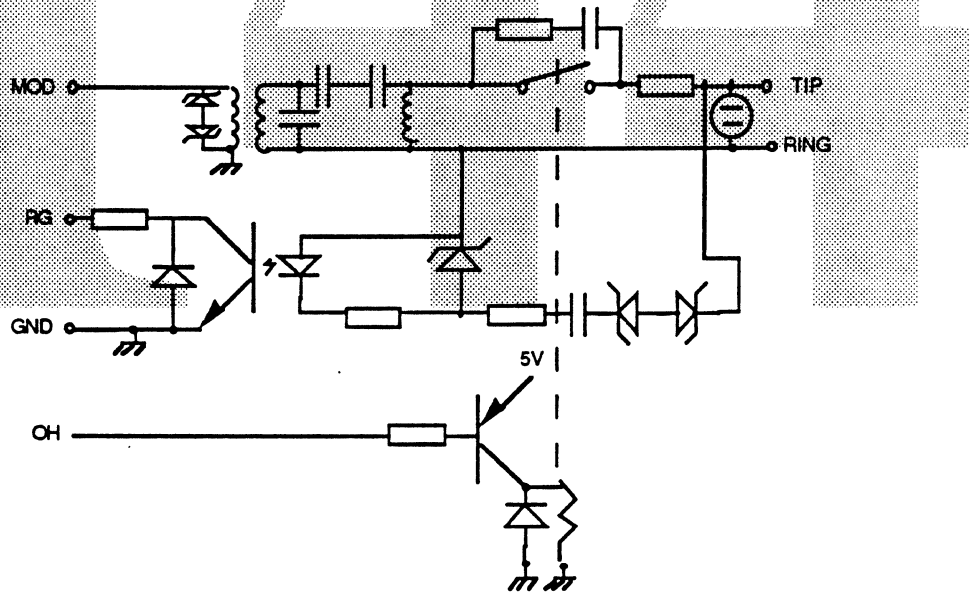
Data Access Arrangement (DAA)

The Data Access Arrangement (DAA) module provides a standardized interface to the various international PTT phone network interconnection standards. Jaguar will use the standard DAAs from Paris that were originally developed for the laptop modem.

The DAA performs the following functions:

- Couples the Jaguar's composite transmit/receive analog data paths from the codec to the standard two-wire subscriber telephone loop[1]
- Signals "ring indication"
- Allows Jaguar to take line "off-hook"
- Provides electrical isolation and transient protection.

The DAA is housed in a small, LocalTalk interface-sized module with pigtail. Jaguar's DAA interface is a small, inexpensive connector (TBD); the other side connects directly to the national phone interface using the specified connector (RJ-11 in the U.S.).



The diagram above shows the guts of the DAA. The MOD signal is the unbalanced voice channel. RG is the ring indication, which pulses on ring signal. OH is the off-hook control, which Jaguar uses to "lift the handset". This signal is also used to pulse dial. This schematic shows a relay actuation for off-hook, but an

[1]the hybrid four-wire to two-wire will be based on the motherboard.

analog switch could be used as well. The only additional components necessary before A/D conversion is the hybrid (two low-cost op amps with simple RC network), lowpass filtering, and gain scaling; the latter two functions are incorporated in the codec chip.

Analog Interface System

The Analog Interface System (AIS) comprises analog-to-digital and digital-to-analog convertors. The AIS is designed to provide high quality signal translation for voice and modem applications, while minimizing the processing requirements for the Jaguar DSP software.

Voice and modem applications place different demands on the AIS. Voice signals are encoded as 8 KHz, 8-bit, mu-law encoded PCM samples. The mu-law logarithmic encoding rule is designed to increase the effective dynamic range (to almost 72 dB) and to improve the signal-to-quantization noise ratio at low amplitudes.

However, this companding algorithm, resolution limit, and sampling rate is unsuitable for practical, high-performance modem emulation. The ideal AIS has programmable sampling rates, with independent sampling rates on both receive and transmit paths. Additionally, a minimum of 14-bit resolution (with guaranteed 10-bit linearity over any 10-bit range) is required. And linear phase antialiasing filters is extremely important.

For this reason Jaguar uses two codecs: one for the voice handset and another, higher performance device for the analog phone connection[1].

Most AIS chips have serial data buses in order to minimize package size and facilitate system interconnection. The AIS will need serial-to-parallel and parallel-to-serial conversion services performed external to the AIS. This can be conveniently and simply performed inside the MAZDA I/O chip.

Analog signals must be lowpassed[2] before conversion to digital in order to satisfy Nyquist's criteria[3] and thus avoid generating aliased signal products in the sampled data stream. This filtering process must be performed by hardware. The filter is commonly realized by active or passive filter networks, switched-capacitor filter chips, or oversampled converters with on-chip filtering. This last approach is preferred, as it provides high quality filtering that can be incorporated in the converter chip. By contrast, discrete filter networks are subject to component tolerances and drift, occupy more real estate, and may require tuning.

The output of the digital-to-analog convertor must be lowpassed as well. This so-called "reconstruction" filter integrates the ADC output, thus removing the "stairstep" artifacts caused by discrete-time sampling. Again, an on-chip switched-capacitor design would be the ideal choice.

There are several possible ways to implement the AIS. The most promising ideas are:

---

[1]the voice handset codec will be incorporated into the ISDN chip if direct handset-motherboard connection is available.

[2]in practice a bandpass filter is used to remove unwanted low-frequency components as well.

[3]The highest significant input frequency must be less than one-half the system sampling rate.

- Extend the 8-bit CODEC found on the AMD 79C30x ISDN chip to 14-bit linear performance, with samples routed through that chip's microprocessor interface[1].
- Use the TI TLC32040 Analog Interface Circuit: a 14-bit ADC/DAC with 10-bit guaranteed linearity over any 10-bit range. This device has the requisite antialiasing and reconstruction switched-capacitor filters on-chip. Sampling rates are software selectable at 7.2, 8, 9.6, and others. Input/output and clocking options are very flexible. This chip is available now.
- Use the Analog Device's ADSP 7801. This is a 16-bit resolution ADC/DAC. This chip also has the required input/output filters. Sampling rates are selectable from any two of 7.2, 8, and 9.6 KHz. This part is just entering fab and first silicon is expected in the February-March 1990 time frame. It is not yet fully spec'ed.
- Use the proposed ITT 3000 stereo codec. The proposed performance far exceeds requirements, but low cost from large volumes could result.

The first option is quite risky and is currently being investigated. Since the analog phone section of the AMD 79C30x occupies about 25% of the die, and is not required in the Jaguar architecture[2], it would be ideal to concentrate this function onto the ISDN transceiver chip.

The second option is the safest, as this part exists now and is fully specified. Also, it has the most flexibility in sample rate selection. This is attractive, as the three following sample rates would be used:

- 7.2 KHz for 300/1200/2400 transmit/receive and V.29 transmitter.
- 8 KHz for voice data: provides easy, higher quality sample rate conversion, particularly in the case of decimation
- 9.6 KHz for v.29 receiver.

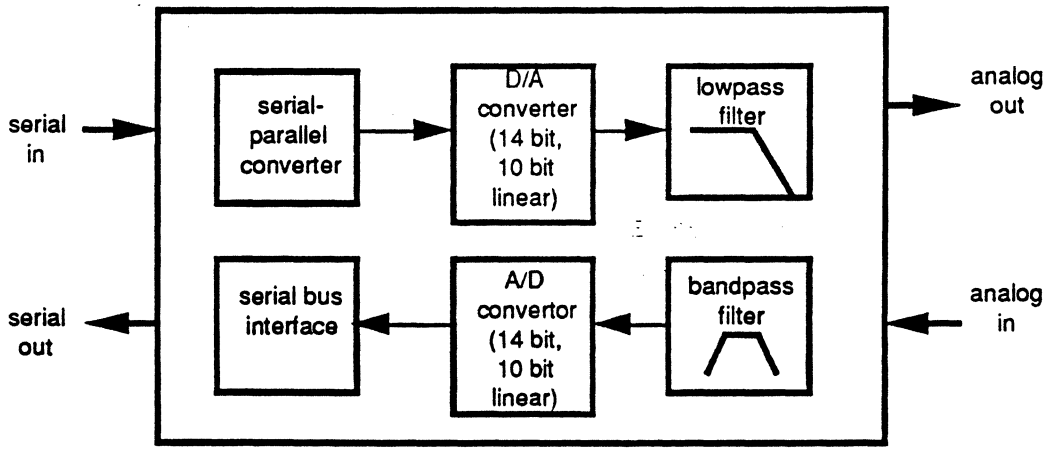
Note that the V.29 FAX modem would ideally like 7.2 KHz transmit sampling but 9.6 KHz receiver sampling: the lower transmit sampling rate reduces the processing requirements without compromising quality. This can be done by the TI part while still maintaining the 8 KHz option.

RALPH assumes that the functions indicated in the diagram below will be provided in a single package. This is the case for the codecs mentioned above. The following discussion refers to the TI 32040, although an alternative codec implementation would operate in a similar manner. The 32040 clock at 4.608 MHz is obtained from the telecom subsystem's master 36.864 MHz crystal, divided by eight. This clock is in turn subdivided internally to the chip to generate its internal 288 KHz filter clock - which is in turn divided by a programmable register to obtain the desired sample rates of 7.2, 8.0, and 9.6 KHz. Note that the filter clock rate is independent of the sample rate: this is required to maintain identical filter characteristics.

---

[1]this option will not be pursued if the handset codec is incorporated into the chip (ref AMD 79C32x).

[2]because A/D and D/A conversion is done locally at the user interface.



The serial bus provides a frame sync pulse which qualifies the data clock. This provides an easy interface to the MAZDA phone I/O module's serial-to-parallel converter.

Automatic Gain Control

The analog signal's amplitude before digital conversion is required for optimal AIS performance. If the analog signal is too low, dynamic range is lost; too high, and serious distortion results. This function is performed by the automatic gain control (AGC) circuit, which is partitioned between hardware and software. The required dynamic range is a function of the codec sensitivity and resolution. Since Jaguar provides floating-point processing, post-conversion normalization is not required except at the symbol demodulation operation.

RALPH monitors the RMS power within the spectrum of interest by scanning the post-conversion DMA buffers. After calculating the signal power, RALPH adjusts the programmable analog gain stage found in the AIS. Some codec chips have programmable attenuators with a dynamic range around 6-25 dB. The upper figure is the lower margin for acceptable AGC operation with telephone signals, whose received power can span a range of 48 dBm (eight bits @ 3dB voltage gain per bit since P = E*E/R).

RALPH modem pseudo-devices

The following modem pseudo-devices are supported (emulated):

V.21 a 300 bps, full-duplex standard using frequency-shift (FSK) modulation. This standard is used outside the USA, most notably in Europe.

103 a 300 bps, full-duplex Bell standard using FSK modulation. This standard is used within the USA

V.22 a 1200 bps, full-duplex standard using phase-shift (PSK) modulation. This standard is used within the USA and is employed by the common contemporary "personal computer" modem - such as the Hayes™ or Apple Personal Modem

212A Bell (American) version of V.22. Uses a slightly different (yet significant) handshake and requires "guard tones" to be transmitted in the answer mode.

V.23 a 600/1200 bps full-duplex standard using phase-shift (PSK) modulation with a 75/150 bps back channel. This standard is used in Europe for videotext applications such as the French minitel.

V.22bis a 2400 bps, full-duplex standard using Quadrature-Amplitude (QAM) modulation. This standard is used both in the USA and abroad.

V.27ter a 2400/4800 bps, half-duplex standard using PSK modulation. This is the low-speed standard for FAX transmission, and is used in Europe for dialup X.25 packet network service as well.

V.29 a 9600 bps, half-duplex standard using QAM modulation. Fallback rates at 7200 and 4800 bps[1] are also specified for use on impaired lines. This standard is universally followed for FAX transmission.

A future implementation will include V.32, a 9600 bps, full-duplex standard using trellis encoded[2] QAM. This standard is extremely compute intensive: an estimated 50% of the XJS processor would be required to emulate this mode.

In short, using intrinsic RALPH data pump, all Jaguars will have the capacity to communicate worldwide with any 300-9600 bps modem, with integral support of group 3 (digital compression) FAX.

Programmatic Interface

RALPH provides "data pump" functionality to the application programmer through its programmatic interface. Great flexibility results from the software-based implementation. RALPH maintains a configuration data structure (the ralph "profile") that defines the current state and operation mode of RALPH's modem pseudo-device.

Normally, the application programmer would configure the RALPH profile directly (this could be stored in a localizable resource). However, a large body of software developers are accustomed to the Hayes™ "AT" command set. In order to provide a measure of compatibility with existing telecommunications software, RALPH's modem interface will also accept the ubiquitous and proven "AT" command set. The "AT" commands will be parsed and RALPH's profile updated accordingly. Several commands may be catenated into a single command string. Common commands (such as dial a number) will be compiled into a macro library to facilitate fundamental programming.

The data pump command set is as follows:

AT Command line prefix **AT**tention. All command strings must be preceded by the **AT**tention command.

---

[1] 4800 bps V.29 uses the V.27*ter* standard

[2] a sophisticated encoding technique that improves modem performance on marginal lines.

A/ Repeat the previous command.

A Set RALPH to answer mode and go offhook immediately.

Dn Dial the number immediately following this command; n is the number to be dialled, which can contain the following embedded dial sub-commands:

- T Tone (DTMF) dialling
- P Pulse dialling (default)
- R Reverse mode: put modem in answer mode after dialling
- W Wait for continuous tone before dialling the next number. RALPH waits for a period of time defined in register S7.
- @ Wait for a quiet answer, that is, one or more rings followed by five seconds of silence. RALPH waits for a period of time defined in register S7.
- , pause the length of time specified by register S8 (default = 2 seconds). Each comma equals the pause length specified.
- ! Flash: place RALPH temporarily on-hook in order to get a new dial tone
- ; return to the command state after dialling. This command can only occur at the end of a dial command string.
- S dial the stored number contained in RALPH profile structure register n.

Hn Controls the switchhook: n = 0 = on-hook (disconnected), n = 1 = off- hook.

In returns RALPH ID/checksum

On puts RALPH in Online data mode. If n = 1, the equalizer retrain sequence is initiated.

Z causes a software reset. RALPH 's configuration profile is set to the default configuration profile.

Bn Set RALPH emulation mode:

0 = 103

1 = V.21

2 = 212A

3 = V.22

4 = V.22bis

5 = V.23

6 = V.29

7 = V.27ter

Mn speaker control. 0 = speaker off; 1 = speaker on when off-hook, speaker off when carrier detected; 2 = speaker always on.

Qn result codes on/off: 0 = codes are sent; 1 = codes not sent.

Sr? read the contents of status register r.

Sr=n Command to set register r to value n.

RALPH configuration profile structure

```
struct {
        UCHAR   emulationMode;          /* what type of modem */
        UCHAR   originateMode;          /* 0 = answer, 1 = originate */
        INT     bps;                    /* transmission rate */
        BOOL    onHook;                 /* phone DAA is on/off hook? */
        BOOL    busy;                   /* called party is busy */
        BOOL    RTS;                    /* caller's ready to send */
        BOOL    CTS;                    /* ralph data pump ready to send */
        INT     operatingStatus;        /* state of ralph's data pump */
        INT     voiceSrc;               /* source of voice data */
        INT     voiceDst;               /* destination of voice data */

union {

struct {
        UCHAR   ringsBeforeAnswer;      /* 0 */
        UCHAR   countRings;             /* 0 */
        UCHAR   escapeCodeChar;         /* '+' */
        UCHAR   crChar;                 /* 0x13 */
        UCHAR   lfChar;                 /* 0x10 */
        UCHAR   bsChar;                 /* 0x08 */
        UCHAR   blindWaitTime;          /* 2 (secs) */
        UCHAR   dialToneWaitTime;       /* 30 (secs) */
        UCHAR   pauseTime;              /* 2 (secs) for comma */
        UCHAR   cdResponseTime;         /* 6 (1/10th sec) carrier detect wait*/
        UCHAR   hangupDelay;            /* 14 (1/10th sec) no carrier to hangup */
        UCHAR   dtmfDuration;           /* 95 (ms) DTMF tones duration/spacing */
        UCHAR   escCodeGuardTime;       /* 50 (1/50th sec) */
        UCHAR   RTStoCTSdelay;          /* 1 (1/100th sec) */
} ATregs;

struct {
        UCHAR   s0, s1, s2, s3, s4, s5, s6, s7, s8;
        UCHAR   s9, s10, s11, s12, s26;
} Sregs;

} regs

}
```

Functions

The following functions represent the beginnings of the lower-level interface to RALPH. These commands are passed to RALPH using asynchronous messages.

SetProfile

```
RalphErr SetProfile(ralphProfile);
ralphProfileStruct * ralphProfile;
```

Description

SetProfile() initializes the RALPH data pump to the operating mode specified by the ralphProfile structure. RALPH will update its operating profile to reflect the passed configuration data and begin operation as specified by the new operating profile.

Arguments

ralphProfile	The RALPH environment profile structure.

Return Value

badCmdErr	The SetProfile() command was issued when a RALPH communication session was active. In this case the caller must terminate the RALPH channel.

See Also

GetProfile

```
RalphErr GetProfile(ralphProfile);
ralphProfileStruct * ralphProfile;
```

Description

GetProfile() returns the ralphProfile structure that defines the current RALPH operating environment.

Arguments

ralphProfile	The RALPH environment profile structure.

Return Value

badCmdErr	The SetProfile() command was issued when a RALPH communication session was active. In this case the caller must terminate the RALPH channel.

See Also

DoCommand

```
RalphErr DoCommand(commandStr, ralphProfile);
char * commandStr;
ralphProfileStruct * ralphProfile;
```

Description

DoCommand () directs RALPH to execute the "AT" compatible ASCII command string. RALPH is in the command state. The following "AT" commands are supported:

Arguments

commandStr	The ASCII-encoded "AT" command string.

Return Value

parseErr	The command string contained an unknown, incomplete, or unimplemented "AT" command.
badCmdErr	A command was issued that is not valid in RALPH's current state: for example, issuing a dial command when online.

See Also

SendData

Syntax

```
RalphErr SendData(dataBuffer, count, framedData);
char * dataBuffer;
int count;
bool framedData;
```

Arguments

dataBuffer	The ASCII-encoded "AT" command string.
count	Number of bytes to be sent.
framedData	boolean flag, indicating that dataBuffer contains ascii encoded data to be framed with start/stop bit(s) according to the current profile.

Description

SendData() transmits the dataBuffer over the RALPH pseudo-device modem. The current operation mode (originate, answer) and emulation mode (modem type) must be already established.

RALPH transforms the digital data buffer into voice-channel modulation for transmission over the phone line. RALPH makes a"best effort" attempt to send the data to the remote consumer, but error detection and recovery is the responsibility of a higher layer protocol.

The framedData boolean, if set, directs RALPH to insert start- and stop-bit framing according to the current RALPH configuration profile. Otherwise, the dataBuffer is transmitted to the remote unaltered.

SendData() is an asynchronous call.

Return Value

noConnectErr	The remote modem has disconnected.

See Also

GetStatus()

ReceiveData

Syntax

```
RalphErr ReceiveData(dataBuffer, count, framedData);
char * dataBuffer;
int *count;
bool framedData;
```

Description

ReceiveData() receives data from the remote. over the RALPH modem. The dataBuffer is filled with up to count bytes. If count is zero, ReceiveData() will return immediately with the actual number of The current operation mode (originate, answer) and emulation mode (modem type) must be already established.

RALPH transforms the digital data buffer into voice-channel modulation for transmission over the phone line. RALPH makes a"best effort" attempt to send the data to the remote consumer, but error detection and recovery is the responsibility of a higher layer protocol.

The framedData boolean, if set, directs RALPH to insert start- and stop-bit framing according to the current RALPH configuration profile. Otherwise, the dataBuffer is transmitted to the remote unaltered.

Return Value

noConnectErr The remote modem has disconnected.

See Also

SPAM: The Signal Processing Access Manager

SPAM - the signal processing access manager - is a low-level task that controls the flow of digital samples inside Jaguar. These samples are generated by the following sources:

- Telephone or ISDN voice channel codec at 8 KHz
- Telephone modem codec at 7.2 KHz or 9.6 KHz
- Hi-fi desktop codec at 48 KHz
- Application software at 8, 22.25, 24, or 48 KHz
- RALPH modem pseudo-devices
- System software (sysbeeps) at 24 KHz

SPAM supervises the WANKEL I/O control program on behalf of its clients. The two major clients of SPAM are RALPH and the Sound Server. SPAM is responsible for managing the multi-buffered DMA buffers (containing digitally-encoded analog data) that are routed between the desktop and telecom (phone line or ISDN) interfaces. SPAM in fact could reside as a member of RALPH's application team, but more likely would be a pink interrupt service request (ISR) routine, owing to the real-time response requirements and minimal processing performed by SPAM.

An efficient multi-buffered DMA structure may be best implemented as part of the RALPH application team. The partition between SPAM and RALPH is made to separate buffer management tasks from the "hard core" data-pump algorithms - hopefully giving RALPH a virtual data stream.

SPAM buffer management

SPAM clients must ensure that their producer/consumer processes are capable of synchronous, real-time operation or DMA underruns will result - with possibly serious consequences. This can cause audio sound "glitches" or loss of synchronization in RALPH modem pseudo-devices.

The DMA buffer size is dictated by SPAM clients. Each application will have its own design centers that determine the buffer size used.

RALPH modem pseudo-devices are very sensitive to the buffer size, as this determines the scheduling granularity. If the buffer is too small, excessive scheduling overhead results, while excessively large buffers make the scheduling epoch too large, which complicates certain algorithms like initial handshaking and software agc.

The typical RALPH buffer size is in the vicinity of 128-1024 bytes, resulting in a DMA completion interrupt from SPAM every 16-128 ms (assuming an 8 KHz sample rate). Conceptually, a much larger buffer could be used (say 10K bytes) once handshaking has been completed. This would provide additional scheduler elasticity.

SPAM channel control program

The SPAM will use a basic WANKEL channel control program to perform multi-buffered I/O to and from the codec and desktop I/O devices. The following simple channel program shows how it might appear. In this picture, an interrupt has just been generated by WANKEL, alerting SPAM that transmit buffer two has just been transferred to the codec that interfaces the phone line to Jaguar. WANKEL has cleared the "ready" (R) bit in the CCW flags for CCW1, indicating that transmit buffer two is empty and thus ownership has returned to SPAM for fill-up. Flag "ready" in CCW0 has been set by SPAM as a result of a previous interrupt/fill-up cycle, transferring ownership back to WANKEL. Since the transmit and receive channels are in synchronous lock-step, receive-channel control program interrupts are superfluous.

CCW	COMMAND	FLAGS	COUNT	ADDRESS
0	codec-tx	R D U I T 1 0 1 1 0	count = 128 - 1024	addr = tx buffer 1
1	codec-tx	R D U I T 0 0 1 1 0	count = 128 - 1024	addr = tx buffer 2
2	branch	R D U I T 0 0 0 0 0	xxx	addr = CCW0
3	codec-rx	R D U I T 1 0 1 0 0	count = 128 - 1024	addr = rx buffer 1
4	codec-rx	R D U I T 0 0 1 0 0	count = 128 - 1024	addr = rx buffer 2
5	branch	R D U I T 0 0 0 0 0	xxx	addr = CCW 3

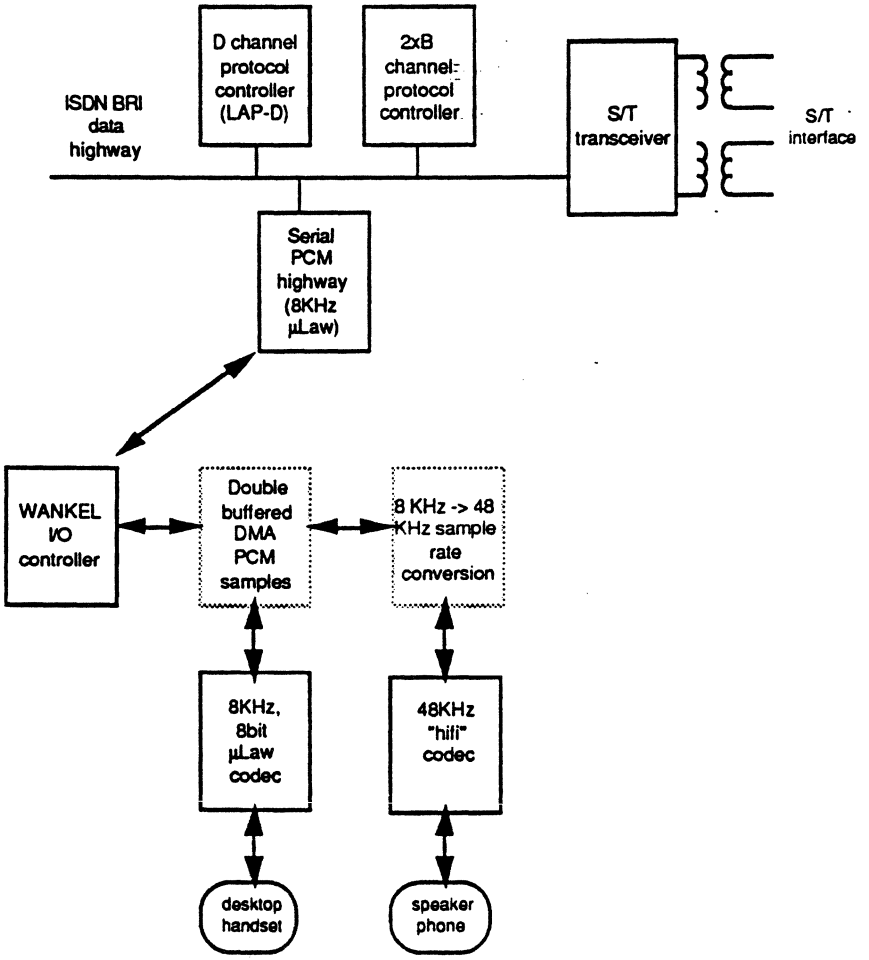
ISDN Basic Rate Interface subsystem

Overview

BRIAN - the ISDN Basic Rate Interface - provides Jaguar with integral voice and date communication facilities as an ISDN "Terminal Equipment 1" operating on the ISDN S/T interface. The BRI allows

connection either directly to the "Network Termination 1" - in which case Jaguar acts as a TE1 on the "Terminal" (T) interface - or as a multidropped TE1 operating on the "System"(S) interface.

BRIAN consists of the following components. A block diagram is shown below.



An S/T interface transceiver that connects the BRI subsystem to the four-wire, 192 KBit/sec, full duplex S/T interface. This transceiver performs complex analog signal processing and collision detection functions. A coupling transformer is required to connect the S/T interface chip to the four-wire line.

A D-channel protocol controller that provides LAP-D[1] (OSI level 2) access to the BRI. This function is often combined with the S/T interface on a single chip. The on-chip controller performs bit-stuffing, flag detection, framing, and address recognition.

Two B-channel protocol controllers for transmission of digital data over the two B channels. Since standard X.25 is used on the B channels, the essential requirement is that the controller handle HDLC frames (such as

---

[1]LAP-D is very similar to the well-known LAP-B used by X.25, with the notable exception that LAP-D has a sixteen bit address field which encodes the level-3 client address.

the SCC). However, ISDN BRI silicon also makes heavy use of serial, time-multiplexed data highways. Thus a choice must be made between using a part like the SCC (using the MAZDA mux routing support), or an alternative chip which provides a direct interface to the serial bus.

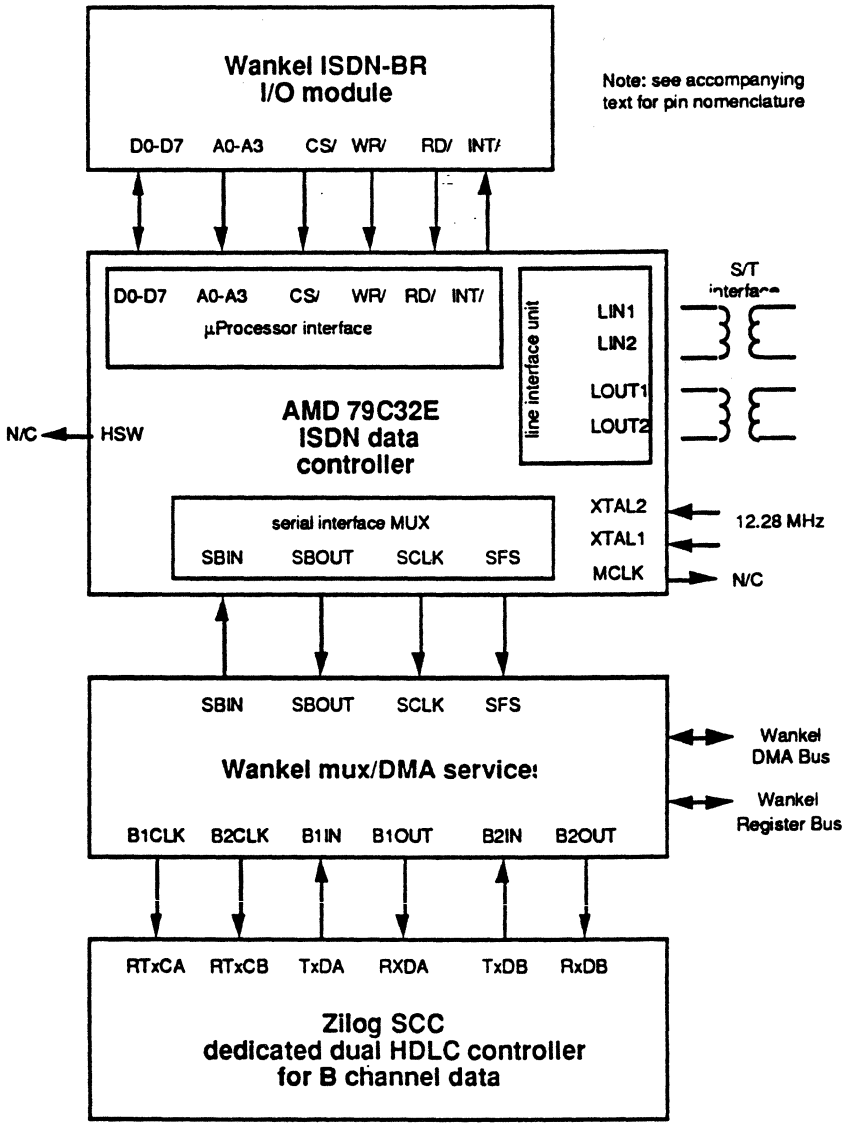
Voice traffic carried on a "B" channel that delivers an 8-bit, mu-law encoded PCM signal at 8,000 samples/second. This data stream is routed to the user interface codec via special mux/demux circuitry inside WANKEL. Two codecs are provided at the user interface Audio signals destined for speakerphone (using the "hi-fi" speakers and standard microphone) are sample-rate converted and sent to the "CD quality" DAC by RALPH.

Implementation

BRIAN uses the AMD 70C32E "ISDN data controller" or IDC, which is an S/T transceiver with "D" channel controller . The IDC consists of the following major components:

- The microprocessor interface(MPI) with 8-bit data bus and three-bit address bus. This simple interface provides direct access to D-, B1-, and B2-channel data buffers, interrupt, command, and status registers. D-channel data is buffered by 16-byte transmit and 32-byte receive FIFOs
- The line interface unit (LIU) which provides physical-level translation service between the serial TDM, four-wire S/T interface as defined by the CCITT I.430 specification. The LIU supports multiframing, and operates in the point-to-point, short- and extended-passive bus configurations.
- A serial multiplexer (MUX) which routes selected data channels between the IDM external serial bus and the LIU.
- Clock generation circuitry. The IDC operates at 12.288 MHz. Stability is specified as ± 800 ppm. The clock is obtained from the telecom subsystem's master 36.864 MHz crystal, divided by three.

ISDN Basic Rate Block Diagrar



IDC Pin nomenclature

- D0-D7: MPI data bus (input/output). Tri-stated if chip not selected
- A-A2: register selection address pins (input)
- CS/: active low chip select
- WR/: active low,specifies write transaction to IDM will take place
- RD/: active low,specifies read transaction to IDM will take place
- INT/: specifies that IDC needs service. Updated every 125µS
- LIN1, LIN2: S/T interface twisted pair input

- LOUT1, LOUT2: S/T interface twisted pair output
- SBIN: serial bus input
- SBOUT: serial bus output
- SFS: serial bus frame sync
- SCLK: serial clock, 192 KHz (3x64KHz)
- XTAL1: N/C
- XTAL2: 12.288 MHz ± 80 ppm clock input
- MCLK: N/C

Wankel mux/DMA services pin nomenclature

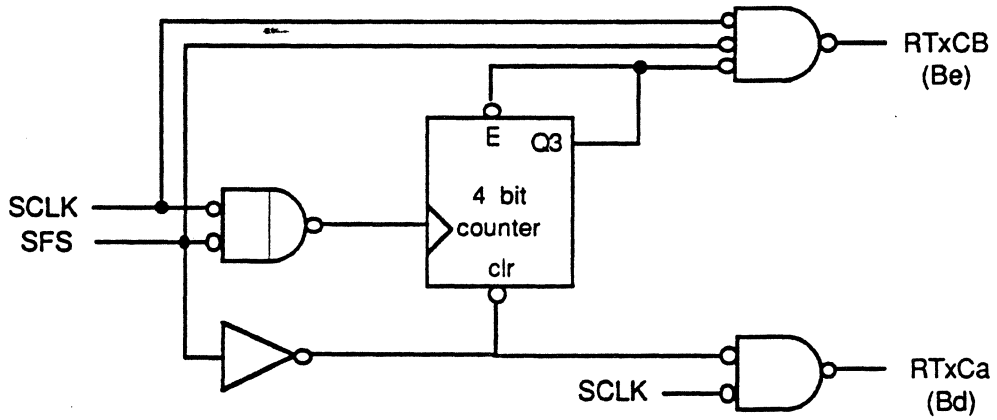
- B1CLK: synchronous clock for tx-rx stream Bb (output)
- B2CLK: synchronous clock for tx-rx stream Bc (output)
- B1IN: IDC Bb data channel (input)
- B2IN: IDC Bc data channel (input)
- B1OUT: IDC Bb data channel (output)
- B2OUT: IDC Bc data channel (output)
- Wankel DMA Bus: internal data highway for B-channel voice data
- Wankel Register Bus: internal program bus

SCC pin nomenclature

- RTxCA: receive clock for Bb serial data bus, sample on positive edge (input).
- RTxCB: receive clock for Bc serial data bus, sample on positive edge (input).
- TxDA: serial transmit data stream Bd = ISDN channel B1 (output)
- TxDB: serial transmit data stream Be = ISDN channel B2 (output)
- RxDA: serial receive data stream Bd = ISDN channel B1 (input)
- RxDB: serial receive data stream Be = ISDN channel B2 (input)

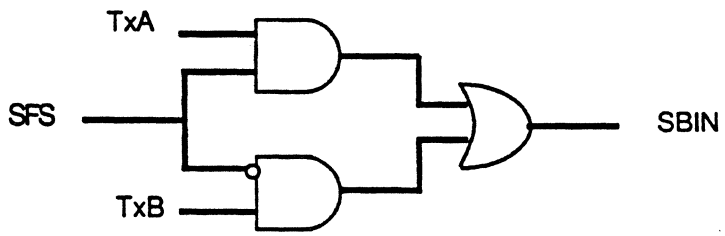
IDC serial bus

The IDC serial bus provides three independent, 64 Kbps serial data channels (called Bd-Be-Bf in AMD literature) that are arranged as a TDM sequence of three octets, one byte from each channel. The ISDN "S/T" interface's "B" channels are routed to the IDC bus channels Bd (ISDN B1) and Be (ISDN B2)using the 79C30's internal, programmable multiplexer. The framing signal SFS marks the beginning of a three byte sequence of logical "B" channels Bd-Be-Bf on the SBIN (serial bus in) and SBOUT (serial bus out) lines. The SBOUT signal will be routed from the IDC to the Wankel engine for demultiplexing. The circuit below, provided for illustration purposes only, should accomplish this trick:

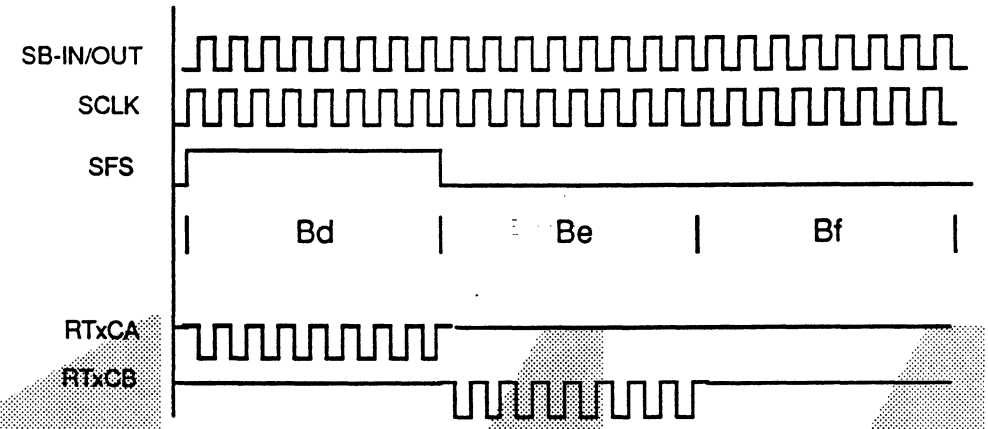


Both B-channels will be routed to a dedicated SCC, operating as a dual-channel HDLC controller to support X.25 traffic using HDLC framing on the B-channels. B-channel data will be routed to and from the SCC using the DMA services provided by the Wankel SCC I/O Module.

Data from the SCC will be multiplexed onto the IDC serial bus using a circuit inside the Wankel. The simple circuit below should suffice:



The resulting Wankel SCC clock signal is thus:



The SBIN signal is applied to both SCC channel A and B data inputs RXA and RXB. Data is synchronously strobed into the SCC at a net rate of 64 Kbps, one byte at a time, on the clock burst present on SCC input/output clocks RTxCA and RTxCB.

The "B" channels also carry PCM voice data (8 KHz, 8-bit μ-law sampled data). This data stream can be routed onto the IDC serial bus Bf channel. See "brian operation" below for more information.

ISDN Standby Power

ISDN power distribution in the US is not regulated by agreement; customers have several options for making their ISDN system operational during power outage - or when the machine is "turned off". The power can be obtained from several sources:

- system power supply on motherboard
- 40 or 48V phantom power, supplied on "S/T" interface
- local wall "brick" with jack[1]

Since the Jaguar XJS processor is required to actually run the ISDN protocol stack, the only feasible alternative is option (1): keep the I/O subsystem powered up, with soft-powerup for the rest of the system. This is exactly what is proposed for the RALPH analog phone service. Options (2) and (3) are probably unfeasible because this power, while perhaps adequate for the ISDN controller circuit, is insufficient to power the entire Jaguar: the central CPU performs ISDN protocol processing, while the speech I/O circuits and keyboard are required for call establishment.

In order to allow the caller to make a call with the machine powered off, we'd need to have the ISDN call management software run on an autonomous card with control processor such as the N&C MCP. If fitted with a SLIC, the user could use a standard telephone set on the desktop.

The entire issue of power feeding is under debate within the ISDN community. A wall "brick" is available from vendors like AT&T that provide power to up to eight ISDN terminals on the "S/T" interface;

[1]used by ATG, for example

however,this "brick" can only power three MCP-based ISDN cards. Power feeding up to the customer in the U.S.: the RBOC "responsibility" ends with the "U" interface termination on the central office side of the NT1 (think of this as the ISDN "phone jack"). Note that the DC-DC converter needed to use the "S/T" power doesn't come cheaply: cost is currently $52 and it measures about 3 inches on a side and 1/4" high![1]

Call answering during "soft" power-off mode would follow the scenario outlined under section "RALPH standby mode". WANKEL's channel control program in the quiescent mode handles only the smallest subset of the entire call processing stack that is necessary to keep the quiescent channel established. This entails recognizing and reflecting an "RR" (Receiver Ready) LAP-D frame over the "D" channel, perhaps every 10 seconds. On reception of a non "RR" frame (presumably a connection establishment request frame or "SABME"), Jaguar will bring itself fully on-line in order to execute the full ISDN call processing protocol stack, which will interpret and process the succeeding frames that lead to the establishment of an incoming call.

BRIAN operation

BRIAN provides two 64 Kbps synchronous "B" data channels and a single 16 Kbps synchronous "D" channel. Call control information is conveyed on the "D" channel using synchronous LAP-D framing. The "D" channel can be used to route packet-switched data as well. MAZDA interfaces to BRIAN using a dedicated I/O module. Under program control, the WANKEL control program writes "D" channel packets through the IDC microprocessor interface, which is buffered through an on-chip 16-byte transmit FIFO. Similarly, WANKEL responds to interrupt requests from the IDC and reads "D" channel frames from the IDC's 32-byte receive FIFO[2].

Full-duplex "B" channel data is routed to the dedicated SCC's two HDLC controllers[3]. The MAZDA mux services generate the required clocks to drive the SCC, providing the concurrent transmit and receive strobes. The SCC is accessed through the MAZDA SCC I/O module - which transfers data to XJS under DMA control.

. An alternative implementation, which would require less software overhead inside MAZDA (if that proves to be an issue) would be to route the PCM voice data to the IDC serial bus channel Bf, with serial-parallel conversion performed by hardware inside the MAZDA chip.

Voice traffic is carried on a "B" channel using mu-law encoded 8-bit PCM at 8,000 samples per second. This data will be read directly on the IDC chip microprocessor interface bus (via an IDC internal register). A WANKEL IDSN control program will move the PCM data under channel program control between IDC chip and the DMA buffers which are accessible to Jaguar's CPU.

SPAM is responsible for routing multi-buffered voice samples between ISDN (or analog telephone via RALPH) and the user interface. Incoming (from BRIAN) samples are processed according to destination. Normally, the voice stream will be routed to the "telephone" codec found in the desktop, which drives the

---

[1] a sub-$30, four square inch converter has been proposed.

[2] the 32-byte receive fifo buffers 160ms of data at 16 Kbps.

[3] X.25 is a CCITT-specified level 2-3 communications protocol on the "B" channel

handset/headset interface. RALPH routes the samples as-is directly to the user-interface bus controller chip under DMA control. The voice samples will be played back using the "telephone" codec. Outbound samples are simply routed from the bus controller chip under DMA control directly to the IDC chip using a WANKEL ISDN channel control program.

Note that a variant of the IDC chip which contains a "telephone" codec is available. If product design determines that Jaguar's design will be of integrated - that is, no high speed serial desktop bus required - then the IDC internal codec will be used and analog voice will be pumped directly between the handset/headset interface and IDC chip[1].

If the hi-fi desktop speakers are selected, RALPH will first convert the samples to linear using table lookup, then upsample to 48 KHz via MAZDA's sample-rate converter. The converted sample stream is then routed directly to the user-interface bus controller chip under DMA control. The voice samples will be played back using the hi-fi codec. Outbound samples (from the user), which arrive under RALPH's control from the 48 KHz codec found on the desktop, are decimated by six in software[2] to create the 8 KHz PCM data stream.

The Jaguar handset is designed for voice-bandwidth operation and thus can be bandlimited to 3.5 KHz. This allows RALPH to decimate the signal by six by simply taking every sixth sample from the user-interface ADC.

BRIAN software will be described in a future document. It will be based on the N&C TELEOS software port to MCP. This software, which was written in "C", operates on the 68K and is currently used to control the Siemens ISDN chip family (S2085 "S/T" interface with 82525 dual HDLC controller and CODEC).

Network

Introduction

The Jaguar Network subsystem comprises the standard components that are found in every machine. The following four facilities are standard on Jaguar:

- Apple "FriendlyNet" LAN interface which provides an Apple AUI to support the Ethernet variants 10 Mbps thick cable, 10 Mbps thin cable, and (subject to further study) "10 Base T" or 10 Mbps twisted pair.
- LocalTalk interface.
- Asynchronous communications.
- Limited synchronous communications.

---

[1]the headset/handset interface uses a standard RJ-14 (mini RJ-11) connector.

[2]hardware support for integer-rate downsampling is under investigation.

FriendlyNet Interface

Jaguar will feature generic motherboard-based support for medium-speed LAN protocols. By incorporating the Apple "FriendlyNet" standard, the use can select among three "flavors" of Ethernet via an external adaptor.

The FriendlyNet LAN adaptor brings "plug 'n' play" simplicity to Ethernet LAN installations. Apple has defined the Apple Attachment Unit Interface (AAUI) that is based on the Ethernet AUI. The Apple AUI is identical to the IEEE 802.3 standard, except that signal pairs are not shielded, and power distribution is designed to operate with existing Apple hardware. The difference between the two AUI standards is as follows:

Apple AUI	Ethernet AUI
+12V @ 200 ma, or +5V @ 375 ma	+12V @ 500 ma
twisted pair transmit	twisted pair transmit w/ground
twisted pair receive	twisted pair receive w/ground
twisted pair collision	twisted pair collision w/ ground
ground	ground

Provisional connector choice is a 14 position, .050" spaced ribbon contact SCSI-2 styleThe Medium Attachment Unit (MAU) is an external "personality module" that interfaces the physical medium to the AUI. The possible alternatives to be supported are:

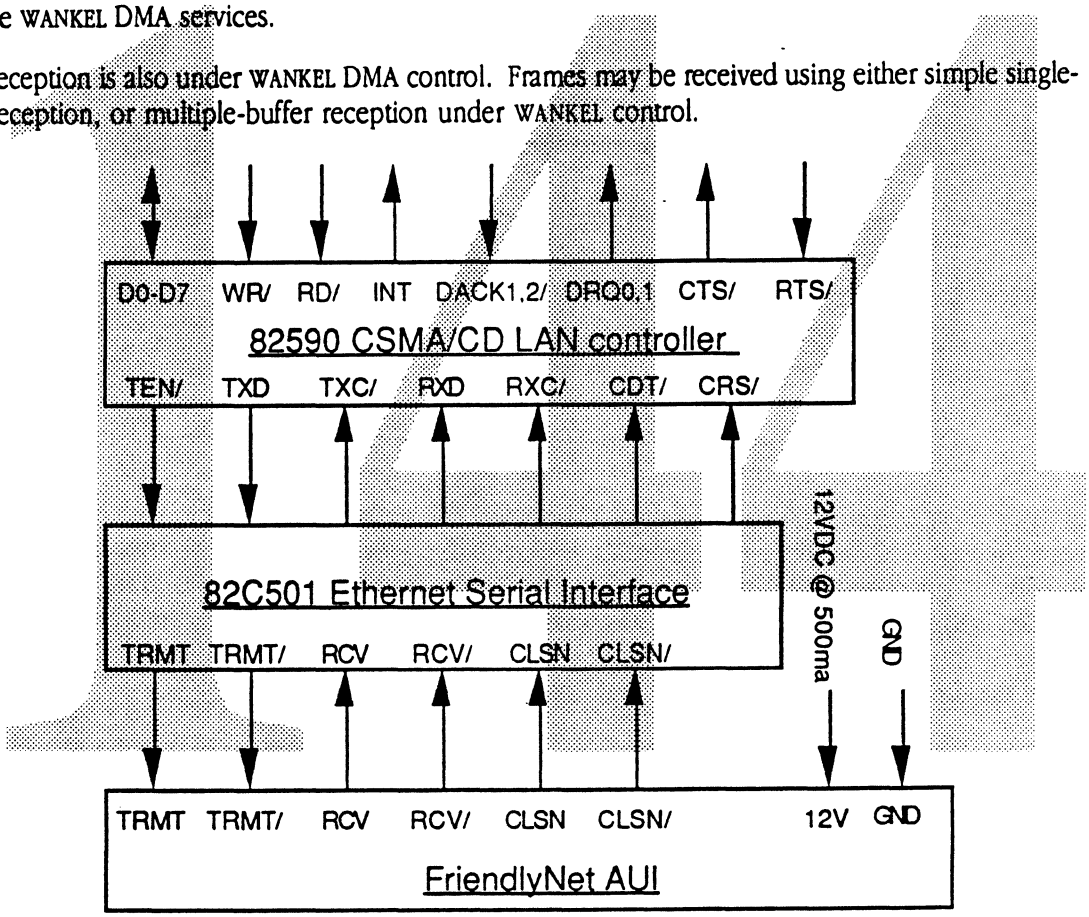
- 10BASE5 - original IEEE 802.3 "Ethernet" specification, using thick (RG-8) 50 Ω cable. Jaguar, which will provide the AUI power (+12V @ 500 ma) specified by IEEE 802.3 , will allow a direct connection from AAUI to an IEEE 802.3 MAU using an AAUI-AUI cable adaptor. The MAUs are available from several third parties. Machines which cannot provide 12V will interface to the IEEE 802.3 MAU using a powered AAUI-AUI converter (under development inside N&C), which provides the specified power via a "wall brick" power tap. Jaguar could use this same convertor, although the passive cable adaptor would be a cheaper solution.
- 10BASE2 - "CheaperNet", thin (RG-58) 50 Ω cable. A "CheaperNet" MAU is currently under development in N&C labs. This will give direct attachment to thin coax via a standard BNC connector.
- 10BASET - Twisted Pair CSMA/CD, twisted pair (22-26 gauge) telephone wire. A 10BASET MAU will be developed within N&C to interface the twisted pair medium to the AAUI. A standard RJ45 snap-in connector attaches the twisted pair to the 10BASET MAU.

There is a great deal of activity in the low-end Ethernet controller market; twisted-pair interfaces and high-integration controllers are under development by most major chip vendors. Devices such as the Intel 82506 Twisted-Pair Ethernet MAU and 82D6 CMOS integrated Ethernet controller/encoder, National SONIC, AMD

ILACC, and other custom, highly integrated components are under investigation. In general, the controller will provide a very simple, slave DMA, 8-bit bus interface to Mazda.

As a reference, the diagram below shows a typical ethernet AUI implementation using the Intel chip set[1]. This chip is similar to the ubiquitous SEEQ 8003. The WANKEL interface uses an eight-bit data path with 64-byte programmable fifos[2]. Frame transmission and reception uses DMA transfers. To transfer a frame, WANKEL prepares a transmit data command block in memory, specifying the preamble, source and destination addresses, length field, and information field. WANKEL then issues a transmit command to the LAN controller interface. The LAN controller then reads the transmit data command block from memory using the WANKEL DMA services.

Frame reception is also under WANKEL DMA control. Frames may be received using either simple single-buffer reception, or multiple-buffer reception under WANKEL control.



Due to the activity in the ethernet chip area, further work with vendors is needed in this area before a selection can be made.

---

[1]Intel ehternet controllers are rumoured to contain variances from IEEE802.3 specifications.

[2]can be configured tx/rx as 48/16, 16/48, or 32/32 bytes.

Asynchronous Interface

This interface provides the standard asynchronous and LocalTalk LAP services found on the Macintosh family of machines. The Jaguar implementation closely follows the Macintosh II variant for compatibility with existing hardware products. These interfaces are controlled by the ubiquitous SCC chip.

In order to support both LocalTalk and async connections on the same port, programmable balanced and unbalanced line transceivers are provided. The interface pinout is identical to the Macintosh "mini-DIN" eight-pin connector:

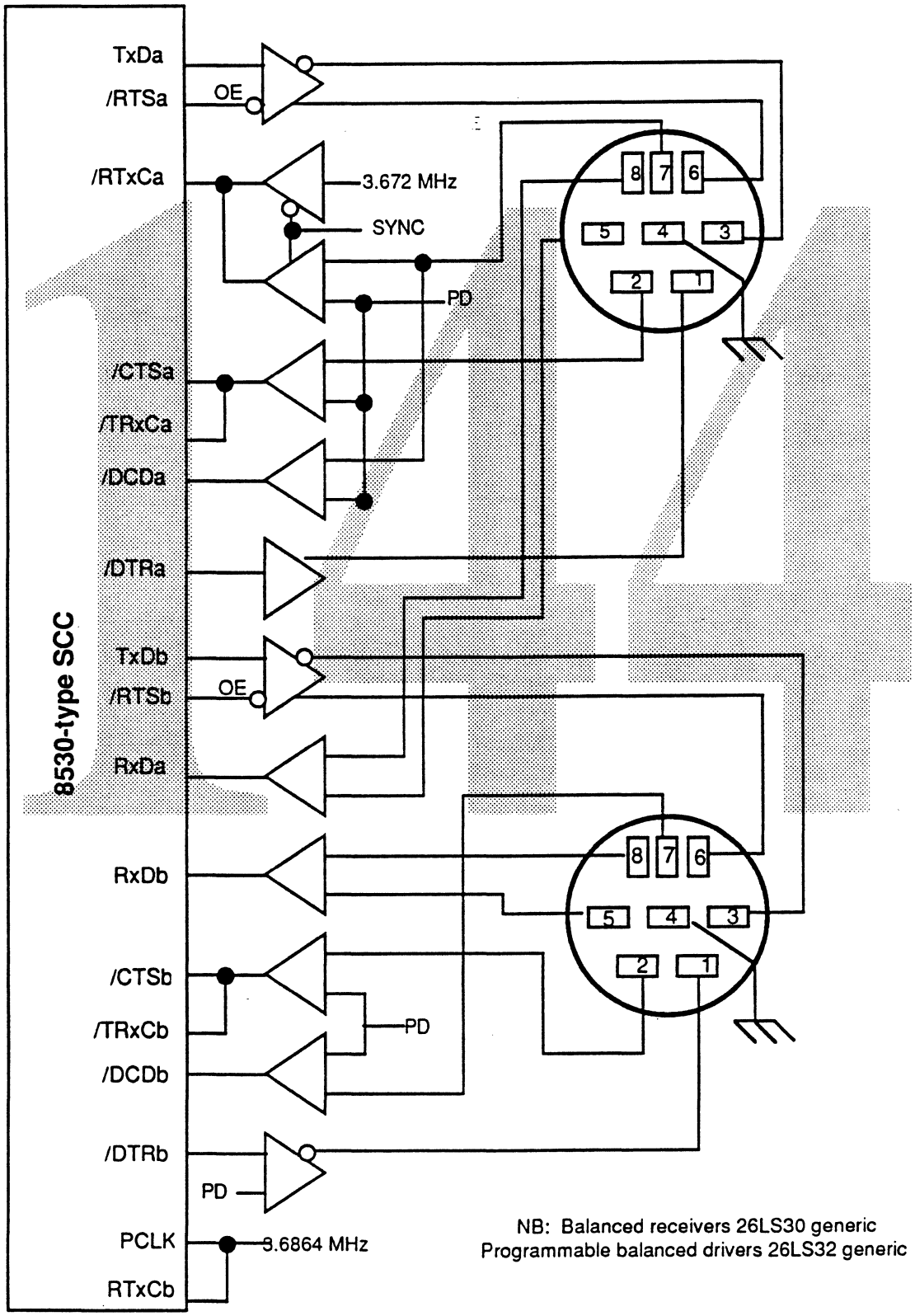
Pin Number	Signal Name	Description
1	HSKo	output handshake
2	HSKi	input handshake or external clock
3	TxD-	transmit data -
4	GND	signal ground
5	RxD-	receive data -
6	TxD+	transmit data +
7	GPi	general-purpose input
8	RxD+	Receive Data +

Since the WANKEL I/O controller shields the SCC from the XJS, a channel control program is required to gain access to generic SCC facilities. One approach is to require a special wankel I/O program for each "custom" appplication. This would yield an efficient solution, but the development time would be excessive. A better idea is to provide a "generic" SCC I/O control program for WANKEL. The generic program provides a limited set of commands that allows a serial driver to read and write the SCC register set. Wankel would interrupt the driver on the SCC "status change" condition.

The diagram below shows a generic hardware implementation[1]. DMA facilities are via the W/REQ pins. Note that full-duplex DMA support will require the DTR/ pin to be implemented in a WANKEL register.

[1]See Macintosh Family Hardware Reference for further details.

Async - LocalTalk - PBX port



NB: Balanced receivers 26LS30 generic
Programmable balanced drivers 26LS32 generic

Programmatic Interface

Normally, a WANKEL control program will be provided for each application as part of the system software release. For example, LocalTalk WANKEL control program would be loaded as part of the system startup tasks onto a port configured for LocalTalk. Since the SCC is shielded from the programmer's address space by MAZDA, there must be a mechanism for "dumb" serial drivers to interface with a virtual SCC. The following four routines are intended to provide a simple, admittedly inefficient access manager interface to a virtual SCC that would be used with the "generic" WANKEL SCC control program. Of course, developers could develop their own WANKEL control programs for special applications.

SendSerialData

Syntax

```
RalphErr SendSerialData(char * dataBuffer, count, portID, async);
    char * dataBuffer;
    int count;
    int portID;
    bool async;
```

Description

SendSerialData() transmits the dataBuffer to the remote channel over the SCC port indicated by portID.

RALPH transmits the data over the serial interface according to the current port configuration profile. Error detection and recovery at the link level is the responsibility of a higher layer protocol.

The async boolean indicates that the call will block the caller until the transfer has been completed.

Return Value

portClosed	The indicated port has not been allocated to the caller.

See Also

ReceiveSerialData

Syntax

```
RalphErr ReceiveSerialData(dataBuffer,  count, portID, timeOut, async);
    char * dataBuffer;
    int *count,
    int portID;
    long timeOut;
    bool async;
```

Description

ReceiveSerialData() receives data from the remote. over the indicated SCC port. The dataBuffer is filled with up to count bytes. If count is zero, ReceiveSerialData() will return immediately with the actual number of The port's configuration profile must be already established.

RALPH transforms the digital data buffer into voice-channel modulation for transmission over the phone line. RALPH makes a"best effort" attempt to send the data to the remote consumer, but error detection and recovery is the responsibility of a higher layer protocol.

The async boolean indicates that the call will block the caller until the transfer has been completed. Asynchronous transfers will be active until count bytes have been read or the timeOut value is exceeded.

Return Value

portClosed	The indicated port has not been allocated to the caller.

See Also

ReadSCCRegs

Syntax

```
RalphErr ReadSCCRegs(SCCRegsPtr, portID, async)
    struct *SCCRegsPtr;
    int portID;
    bool async;
```

Description

ReadSCCRegs() updates the passed in SCC register structure to reflect the state of the physical SCC register set on the indicated port .

The async boolean indicates that the call will block the caller until the request has been completed.

Return Value

portClosed	The indicated port has not been allocated to the caller.

See Also

WriteSCCRegs

Syntax

```
RalphErr WriteSCCRegs(SCCRegsPtr, portID, async)
    struct *SCCRegsPtr;
    int portID;
    bool async;
```

Description

WriteSCCRegs() updates the SCC registers set on the indicated port to the values passed in SCC register structure.

The async boolean indicates that the call will block the caller until the request has been completed.

Return Value

portClosed The indicated port has not been allocated to the caller.

See Also

LocalTalk Interface

A LocalTalk interface (230.4 Kbps) will be standard on the machine. A WANKEL I/O control program performs the AppleTalk LAP services traditionally done by the CPU or PIC chip. Programmable balanced/unbalanced transceivers are required for combining LocalTalk and async services on the same port. The 26L30/26L32 transceiver pair traditionally fill this rôle.

LocalTalk LAP packets are read and written by the wankel control program without intervention from the user. The following commands are defined (for compatibility with existing applications:

WriteLAP(WDSstructure) - write a LocalTalk LAP frame onto LocalTalk medium.

AttachPH(ALAPtype, protoHandler) - attach the protoHandler to LocalTalk protocol table Incoming frames matching the ALAPtype will be received and passed to the caller.

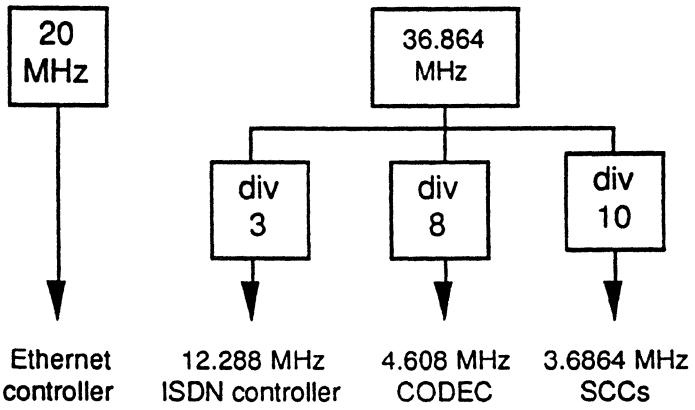
DetachPH(ALAPtype, protoHandler) remove the protocol handler from the protocol table.

PBX / sync modem interface

SCC port A can be configured to accept an external receive/transmit clock. This feature will be used by third parties to provide high performance PBX interconnect boxes using synchronous protocols at bit rates sufficient for concurrent PCM voice and data traffic. This channel interfaces to WANKEL under hardware DMA control.

Network/Telecom clocks

The diagram below shows the clocking requirements for the network/telecom devices described in this document. Since the Ethernet controller derives the 10 MHz Ethernet clock by internally dividing the 20 MHz clock by two, there is no duty cycle restriction on the clock waveform.



Section 6

Sound facilities

I/O Section

Hardware

Output

The desktop will contain a 16-bit stereo digital-to-analog converter (DAC) running at 48 kHz enabling CD-quality output, and a 8-bit mono mu-law DAC running at 8 kHz enabling telephone-quality output (figure 5.1). Both DACs are controlled by serial interfaces that will be connected to MAZDA or to a desktop ASIC. Three of Mazda's output DMA channels move synchronous data from system RAM to the DACs.

The stereo 16-bit DACs along with programmable output attenuation (0 to -22.5dB in -1.5dB steps), a 2-bit input port and a 2-bit output port are included in the ITT UAC 3000 CODEC. The output volume and destination of the 16-bit DACs will be software selectable by the user and by applications via the programmable output attenuator and the I/O ports of the UAC 3000. This information is encoded in the serial data stream sent to the UAC 3000 over the DIN pin. Up, down and mute volume keys are needed on the keyboard for quick real-time adjustments of the output volume.

The DAC's output will be sent through a power amplifier to a pair of internal speakers that are facing the user. This output can be disabled by modifying the output ports of the UAC 3000.

A line level stereo output is supplied with a stereo subminiature phone jack, located in the rear of the desktop. This will allow the user to connect to a stereo or amplified speakers. Insertion of a plug into the line level output jack is detected by an input port on the UAC 3000. This allows the software to disable the internal speakers when detecting the insertion of a plug into the external line level output.

A stereo subminiature phone jack, located on the left side of the desktop, will allow the user to connect the output to either headphones or a high quality headset. A headphone level output is achieved by sending the DAC's signals through a gain section via a pair of low distortion Op Amps. This output is always enabled, not forcing the user to unplug the headphones to enable the speakers, much like that of consumer stereos.

An RJ-14 phone jack, located on the left side of the desktop, will allow the user to connect the 8-bit mu-law output to either a telephone handset or headset. The additional CODEC enables the user to maintain high-quality stereo output at the same time he/she is using the telephone handset.

Input

The desktop will contain a 16-bit stereo analog-to-digital converter (ADC) running at 48 kHz enabling CD-quality input, and an 8-bit mono mu-law DAC running at 8 kHz enabling telephone-quality input (figure 5.1). The ADCs supply data to MAZDA or to a desktop ASIC via a serial interface defined below. Three of Mazda's input DMA channels move Synchronous data from the ADCs to system RAM.

The stereo 16-bit ADCs along with programmable input gain (0 to 22.5dB in 1.5dB steps) and multiplexed inputs are included in the ITT UAC 3000. The input level and source to the 16-bit ADCs will be software selectable by the user and applications via the programmable input gain and the multiplexed input of the UAC 3000. This information is encoded in the serial data stream sent to the UAC 3000 over the DIN pin.

One channel of the ADC is connected to a voice quality directional microphone placed in the front of the desktop. A microphone level input is achieved by sending the microphone's output through a 60dB gain section via a low distortion Op Amp.

A line level stereo input is supplied with a stereo subminiature phone jack, located in the rear of the desktop. This will allow the user to connect to a CD player, tape deck, or external pre-amped microphones. Insertion of a plug into the input jack is detected by an input port on the UAC 3000. This allows the software to disable the internal microphone when detecting the insertion of a plug into the external line level input.

An RJ-14 phone jack, located on the left side of the desktop, will allow the user to connect the 8-bit mu-law input to either a telephone handset or headset. The additional CODEC enables the user to maintain high-quality stereo input at the same time he/she is using the telephone handset.

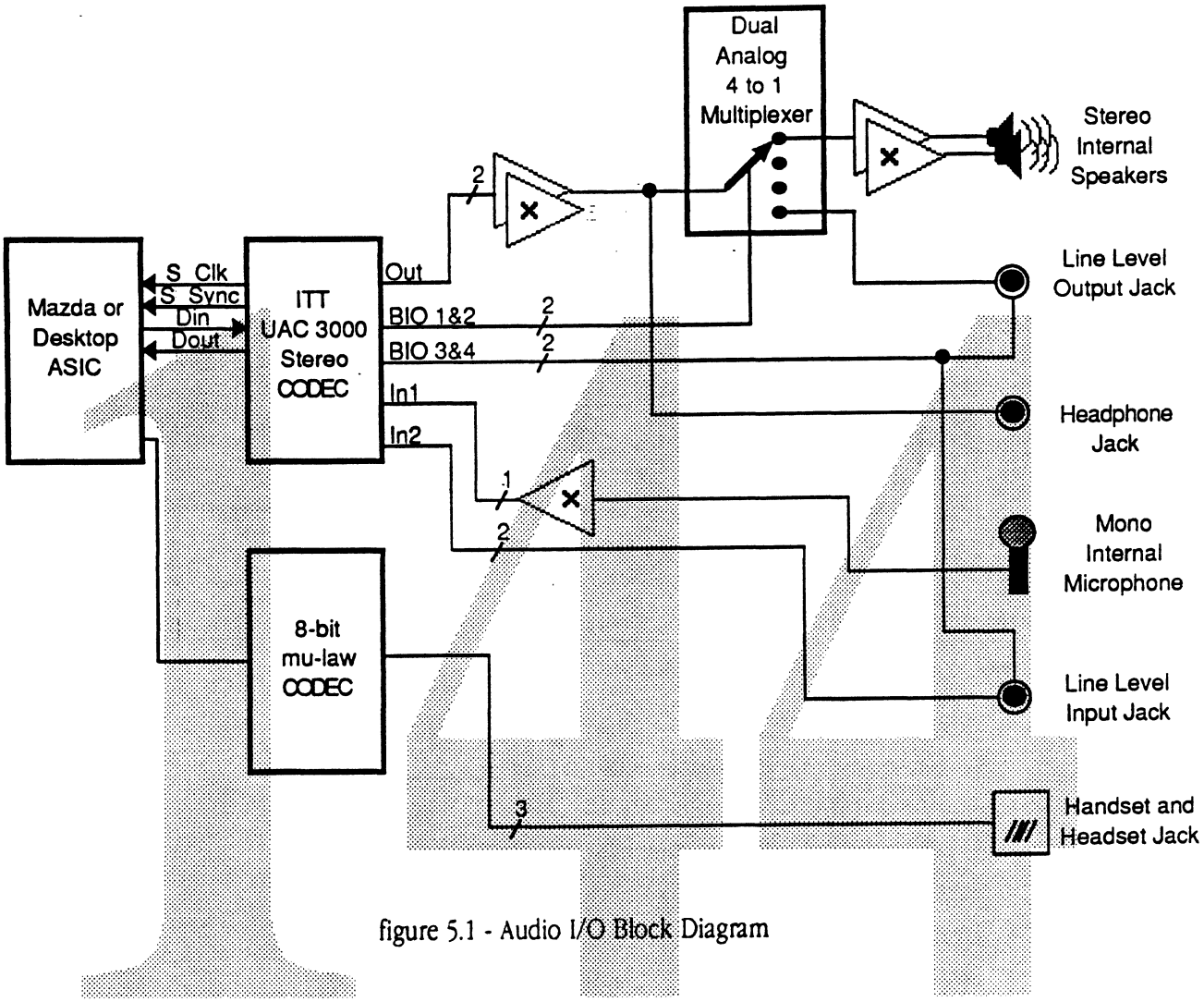


figure 5.1 - Audio I/O Block Diagram

ITT UAC 3000 Stereo CODEC

The UAC 3000 is a stereo ADC and DAC currently being developed by ITT for Apple computer as a low cost solution for high quality audio. The UAC 3000 has a conversion rate up to 48 kHz and a resolution of 16-bits in and out. The UAC 3000 has a 4 wire serial interface that is currently being defined by Apple Computer. Very few external analog components are required to support the UAC 3000, due to internal anti-aliasing input filters and internal oversampling output filters.

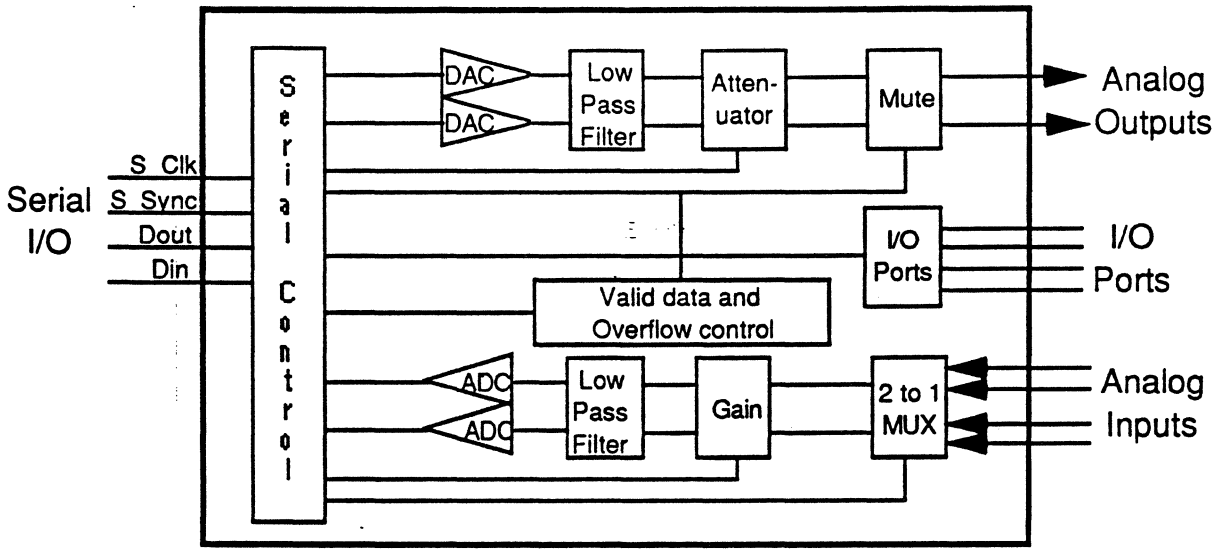


figure x.x - UAC 3000 Block Diagram

Features

The UAC 3000 contains the following features:

- Stereo 16-bits
- Programmable gain input amplifiers
- Sigma-delta ADC
- 3rd order moving time averaging filter on the input
- Post ADC conversion filter
- Oversampling filter/Noise shaper on the output
- Rotating current source DAC
- Programmable output attenuator
- 80dB S/N & .04% THD from ADC input to DAC output
- 4-bit Bi-directional port
- 2 position multiplexed input
- mute
- power down mode
- low cost

Pinout

1 Mode Select (0 => ITT bus, 1 => Apple bus)
2 S_Sync
3 S_Clk
4 Dout

5 Din
6 Clock
7 L_In1
8 L_In2
9 R_In1
10 R_In2
11 L_Out
12 R_Out
13 Analog Gnd
14 Analog Vdd
15 Vss_Digital
16 Vdd_Digital
17 Power_Down (0 => power down, 1 => operate)
18 BIO_1 (bit I/O #1)
19 BIO_2 (bit I/O #2)
20 BIO_3 (bit I/O #3)
21 BIO_4 (bit I/O #4)
22 Group Select 1
23 Group Select 2

Serial Interface

The following is the currently proposed 4 line serial interface. (This proposal will be finalized by 1st quarter, 90)

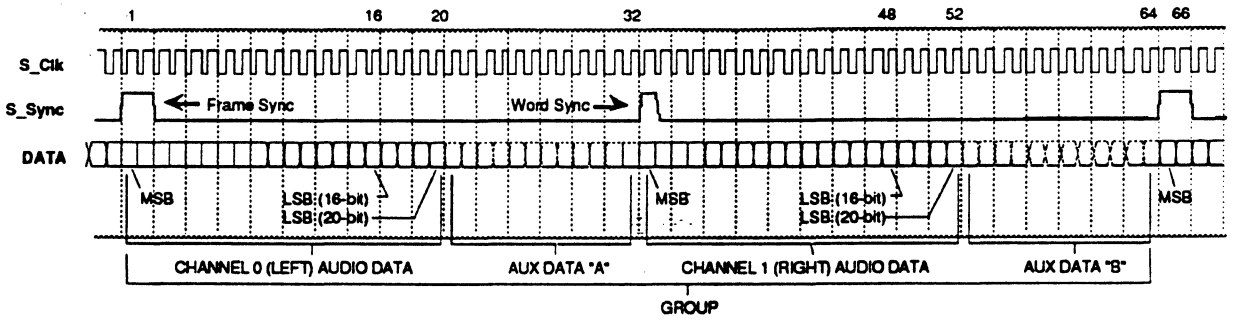
- **S_Clk:** This signal is used to clock serial data in and out of the UAC3000. This clock runs at 64X the sample rate and is generated in the UAC 3000. The negative transition is for data change and the positive edge is for data sampling.
- **S_Sync:** This signal is used to indicate the start of a word or a frame. The frame sync indicates the start of the left channel data and the word sync indicates the start of the right channel data. The line is normally low, and goes high for one bit cell at the beginning of a word/frame boundary (MSB, cell 1). The next cell is also one for a frame boundary, or zero for a word only boundary. This signal is supplied to the UAC 3000.
- **DIN:** Serial data from the ADC's in a 2's complement format MSB first.

Cells	Data
1-16	Left ADC data MSB first
17-20	0 (future expansion to 18 & 20-bit ADCs)
21	Expand (currently 0; for future expansion)
22	ADC valid data
23-24	Left and Right channel ADC overflow

25-28	Error number
29-32	Revision #
33-48	Right ADC data MSB first
49-52	0 (future expansion to 18 & 20-bit ADCs)
53-56	Input Ports
57-64	future bits (currently 0)

- **DOUT:** Serial data to the DAC's in a 2's complement format MSB first.

Cells	Data
1-16	Left DAC data MSB first
17-20	0 (future expansion to 18 & 20-bit DACs)
21	Expand (currently 0; for future expansion)
22	Mute
23-24	Input Select (controls the Input MUX)
25-28	Left input gain (0 to 22.5dB in 1.5dB steps)
29-32	Left output attenuation (0 to -22.5dB in -1.5dB steps)
33-48	Right DAC data MSB first
49-52	0 (future expansion to 18 & 20-bits)
53-56	Output Ports
57-60	Right input gain (0 to 22.5dB in 1.5dB steps)
61-64	Right output attenuation (0 to -22.5dB in -1.5dB steps)



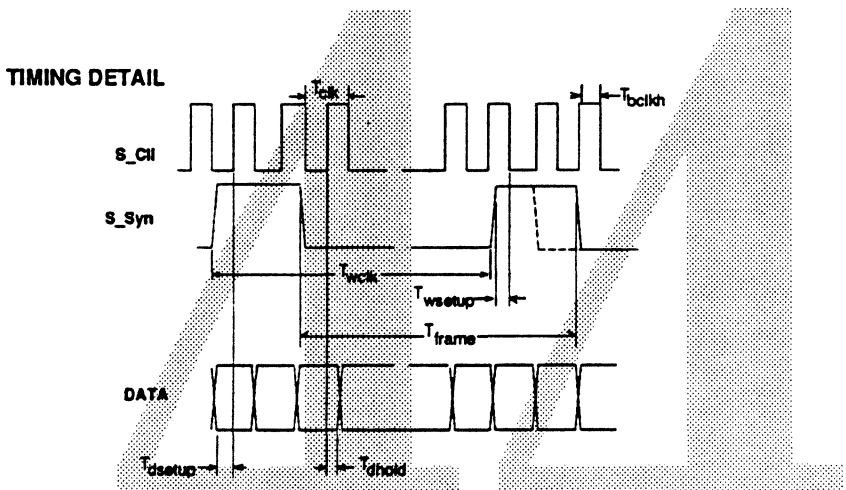


figure 5.2 - UAC 3000 Timing Diagram

Software

Introduction

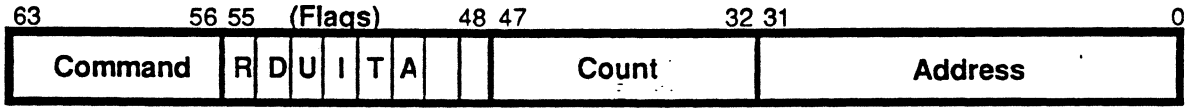
The audio I/O section includes five DMA channels:

- Two synchronous input DMA channels supplying data to the high-quality stereo DACs.
- Two synchronous output DMA channels receiving data from the high-quality stereo ADCs.
- One DMA channel to send and receive status from the ITT 3000.

Each DMA channel is controlled by a Channel Program (CP). A CP contains a data structure which describes a series of I/O operations to be performed. A Channel Control Word (CCW) is a 64-bit word that describes each individual I/O operation (figure 5.3). After a CP is defined by the XJS, a pointer to its location is sent to Mazda which begins the DMA transfer (for more information on CP's refer to the Mazda I/O architecture).

I/O Section Channel Control Word Definition

The following defines the CCW for the audio I/O section:



bits	Flags
55	Ready (0=XJS;1=Mazda)
54	Data Chain
53	Update
52	Interrupt
51	Timer Int
50	Aborted

Commands	
DAC Data (read)	0
ADC Data (write)	1
UAC 3000 Write Status	2
UAC 3000 Read Status	3
Jump	255

figure 5.3 - Audio I/O Section Channel Control Word Definition

- **DAC Read:** This command specifies the location and quantity of 16-bit linear 2's complement audio data to be DMAed to each of the UAC 3000's DACs.
- **ADC Write:** This command specifies the location and quantity of 16-bit linear 2's complement audio data to be DMAed from each of the UAC 3000's ADCs.

- **UAC 3000 Write Status:** This command specifies the location of the data to be written into the UAC 3000's write status registers.

  Shown below is the format and definition of the data pointed to by write status pointer:

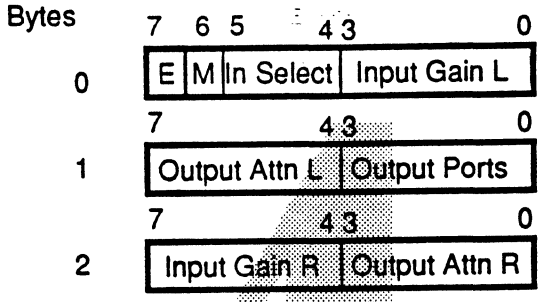


figure 5.4 - UAC 3000's Write Status Format

UAC 3000's Write Status Data Definition

- **Expand (E):** This bit can be used in the future to increase the number of write status registers. Currently it must be a zero for future compatibility.
- **Mute (M):** A value of one will mute the output of the DACs. Value of zero for normal output.
- **Input Select:** This value selects between the four position of the input multiplexer, located in the UAC 3000 at the input of each of its ADCs.

Values	Position
0	Internal Microphone
1	Line Level Input
2-3	Undefined

- **Input Gain Left & Right:** This 4-bit values adjusts the programmable input amplifiers, located in the UAC 3000 at the input of each of its ADCs.

  The following chart defines the gain values:

Values	Gain (dB)
0	+0.0
1	+1.5
2	+3.0
3	+4.5
4	+6.0
5	+7.5
6	+9.0

7	+10.5
8	+12.0
9	+13.5
10	+15.0
11	+16.5
12	+18.0
13	+19.5
14	+21.0
15	+22.5

- **Output Attenuation Left & Right**: This 4-bit values adjusts the programmable output attenuators, located in the UAC 3000 at the output of each of its DACs.

  The following chart defines the gain values:

Values	Gain (dB)
0	+0.0
1	-1.5
2	-3.0
3	-4.5
4	-6.0
5	-7.5
6	-9.0
7	-10.5
8	-12.0
9	-13.5
10	-15.0
11	-16.5
12	-18.0
13	-19.5
14	-21.0
15	-22.5

- **Output Ports**: This value modifies the two output ports included in the UAC 3000. The port controls an analog 4 to 1 multiplexer that selects the audio output destination.

Values	Destination
0	Internal Speakers
1	off
2	off
3	Line Level Output
4-7	Undefined

- **UAC 3000 Read Status:** This command specifies the location to dump the contents of the UAC 3000's read status register.

  Shown below is the format and definition of the data pointed to by read status pointer:

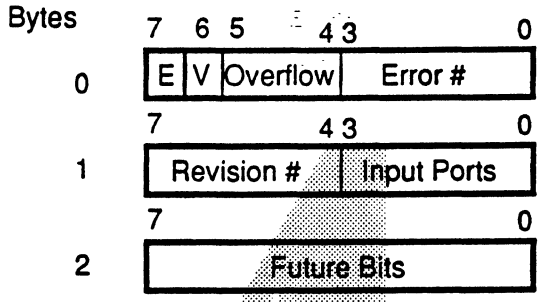


figure 5.5 - UAC 3000's Read Status Format

UAC 3000's Read Status Data Definition

- **Expand (E):** This bit can be used in the future to increase the number of read status registers. Currently it is read as a zero.
- **ADC Valid Data (V):** A value of one indicates valid ADC data. Value of zero for invalid data. This is used to indicate that the ADC has completed initialization following power up or low power mode, and is generating valid data.
- **ADC Overflow:** Indicates that clipping is occurring in the ADC conversion process. (bit 4 for left & bit 5 for right)
- **Error Number:** These bits should all be zero unless an error has occurred. (to be defined)
- **Revision Number:** Indicates the feature set available from this Codec. Currently read as zero.
- **Input Ports**: These 4-bits are modified by the input ports included in the UAC 3000. The MSB 2-bits are connected to the line level input jack (bit 3) and the line level output jack (bit 2). A bit level of one indicates that a plug is inserted into the jack.
- **Future Bits:** Extra bits that can be used for future features. Currently read as zero.

- **Jump:** The jump command causes MAZDA to update its current CPP to the location specified by the address value and continue executing CCWs from this location.

Figure 5.6 shows an example of a Channel Program used for downloading data to each of the UAC 3000 DACs. In this example there are three 5 ms buffers. After each buffer is emptied by Mazda, an interrupt is sent to the XJS. It is then the responsibility of the Sound Server to refill the buffer and set the ready flag before Mazda has returned to the buffer (i.e. 10 ms).

63 Cmnd 56	55 Flags 48	47 Count 32	31 Address 0
DAC Data	1 1 1 1 0 0	480 (5ms)	DAC Data Pointer (1)
DAC Data	1 1 1 1 0 0	480 (5ms)	DAC Data Pointer (2)
DAC Data	1 1 1 1 0 0	480 (5ms)	DAC Data Pointer (3)
Jump	1 0 0 0 0 0	Undefined	Top of CCW Pointer

figure 5.6 - Audio Output Channel Program

Figure 5.7 shows an example of a Channel Program used for uploading data from each of the UAC 3000 ADCs. In this example there are three 5 ms buffers. After each buffer is filled by Mazda, an interrupt is sent to the XJS. It is then the responsibility of the sound server to empty the buffer and set the ready flag before Mazda has returned to the buffer (i.e. 10 ms).

63 Cmnd 56	55 Flags 48	47 Count 32	31 Address 0
ADC Data	1 1 1 1 0 0	480 (5ms)	ADC Data Pointer (1)
ADC Data	1 1 1 1 0 0	480 (5ms)	ADC Data Pointer (2)
ADC Data	1 1 1 1 0 0	480 (5ms)	ADC Data Pointer (3)
Jump	1 0 0 0 0 0	Undefined	Top of CCW Pointer

figure 5.7 - Audio Input Channel Program

Figure 5.8 shows an example of a Channel Program for reading and writing to the internal register of the UAC 3000.

63 Cmnd 56	55 Flags 48	47 Count 32	31 Address 0
UAC 3000 Write Status	1 0 1 0 0 0	3	UAC 3000 Write Status Pointer
UAC 3000 Read Status	1 0 1 1 0 0	3	UAC 3000 Read Status Pointer
Jump	1 0 0 0 0 0	Undefined	Top of CCW Pointer

figure 5.8 - UAC 3000's Status Channel Program

Sample Rate Converter (SRC)

Introduction

Sample rate conversion is the process of changing the sample rate of a set of data representing a given signal into another set of data representing the same signal at a different sample rate. In its general form, the problem is to compute signal values at arbitrary times from a set of discrete samples. The problem is therefore one of interpolation between samples of the original data.

Conceptually, this process can be thought of in three stages. First, an oversampled signal is achieved by inserting samples of zero value at the oversampled sample rate. The new signal is then passed through a filter to give the extra samples real values. The filter's task is to prevent images of the input sampling spectrum from appearing in the output spectrum. The filter necessary is a finite impulse response (FIR) low-pass filter with a response which cuts off at the Nyquist frequency of the input sample rate, in the case of up sampling, and the output sample rate, in the case of down sampling. The third stage is to decimate the signal down to the desired output sample rate.

Sample rate conversion is required for several reasons:

- If the sound file has a rate lower from that of the output hardware (48 kHz), its sample rate must be converted before being sent to the DAC. Lower sample rates are very common due to the increased storage and processing efficiency. This also gains us compatibility with Macintosh (22 kHz) and other Hardware file formats.
- To shift the perceived pitch of a sound. For example, if a piano is sampled at 24 kHz and then played back at 48 kHz the perceived pitch would be an octave higher.
- To route digital signals between the high quality audio I/O system(48 kHz) and the phone, ISDN and Handset (8 kHz).
- When sampling a sound from a constant rate ADC (48 kHz), it is often desirable to lower the sample rate to compress the data and improve computational efficiency for DSP algorithms at the expense of decreasing the frequency bandwidth.

Since adequate quality multi-voice sample rate conversion requires a large percentage of the CPU bandwidth (figure 5.8), we have included a dedicated variable order RAM to RAM sample rate converter as a subsection of Mazda.

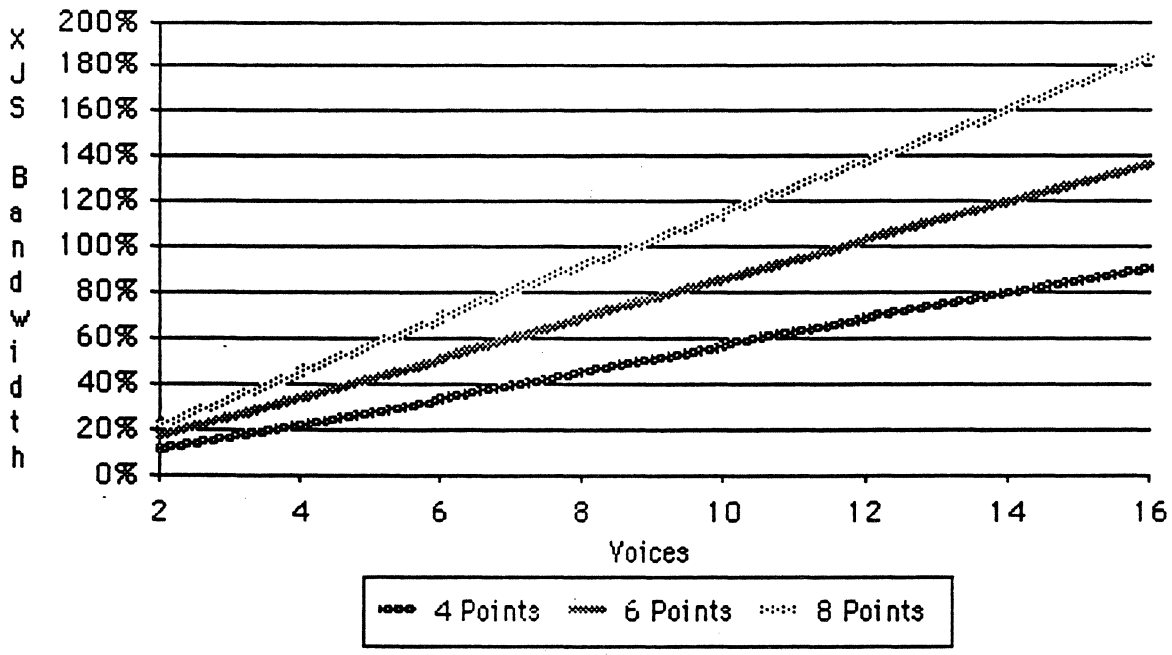


figure 5.8 - CPU Bandwidth vs. Channels & Quality

Dataflow

Figure 5.9 shows the basic flow of data through the sample rate converter. First, incoming samples are DMAed into a RAM FIFO. For every sample in the FIFO a coefficient is calculated by linear interpolating between two adjacent locations in the coefficient RAM. An interpolated sample is calculated by summing the products of the FIFO samples multiplied by the calculated coefficients. The interpolated sample is then sent through a gain section to either attenuate or amplify the output signal. Finally the interpolated sample is limited to a signed 16-bit number before being DMAed out to RAM.

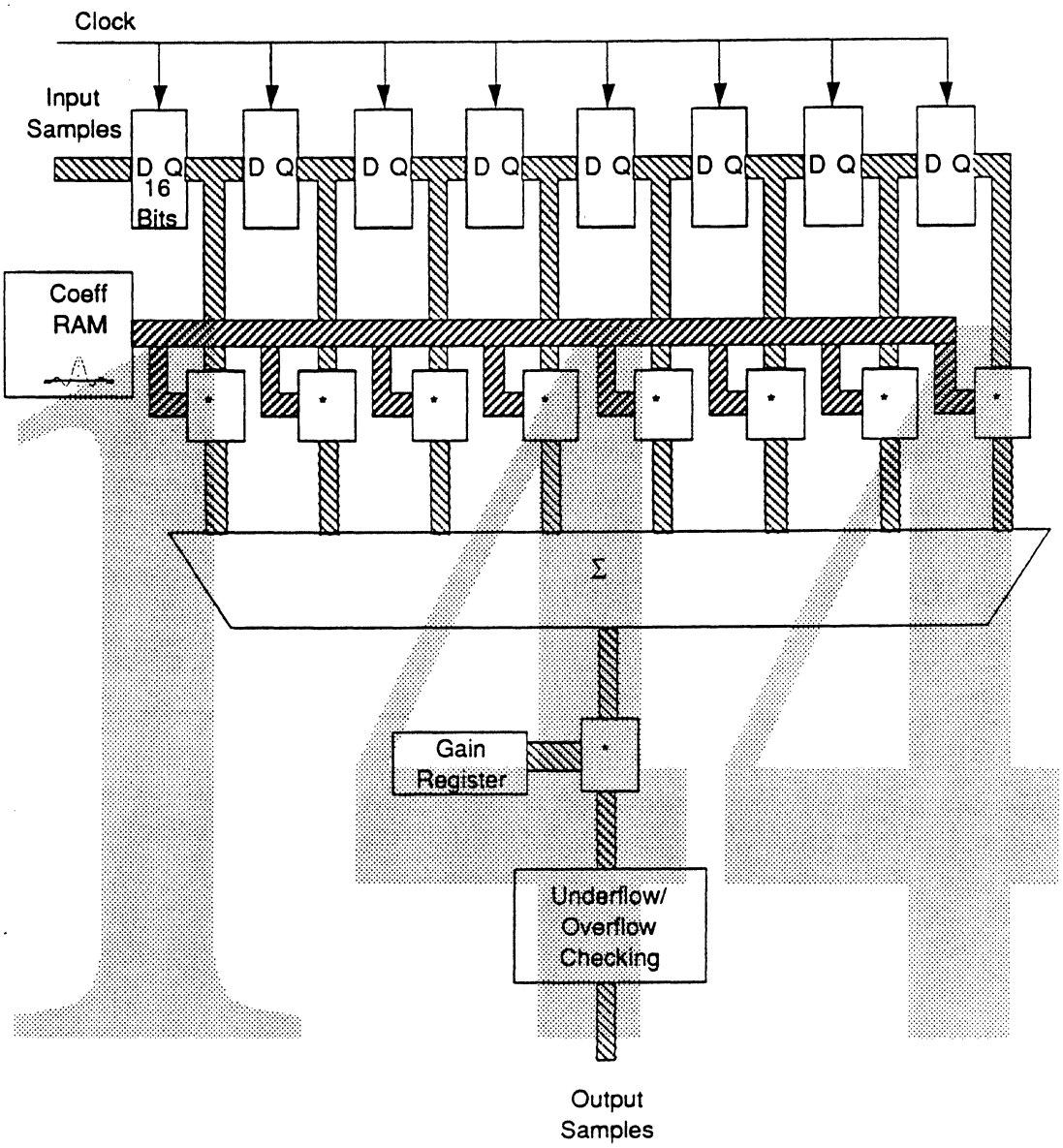


figure 5.9 - Sample Rate Converter Block Diagram

Features

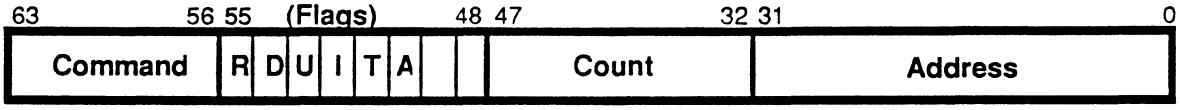
- Variable order The SRC allows us to vary the order of the FIR filter from one to sixteen points by adjusting the numofLobes register. Variable order allows us to decrease the order of the FIR filter to trade-off quality of individual sounds for a higher quantity of sounds. This is not a bad trade-off considering that the decrease in quality will be hidden by the complexity of the final output, resulting from the quantity of different sounds being played.

- RAM Coefficients Since the 64 word coefficient table is stored in RAM and loaded every frame, the filter characteristics can change every frame. This allows the user to change the low pass filter cutoff frequency in order to support variable order filters and multiple down sampling ratios.
- Linear interpolation on/off By modifying the lerp flag the user can turn linear interpolation of the coefficient table on or off. Linear interpolation places a zero at the output Nyquist frequency, which effectively improves the stop-band rejection of the FIR filter. Turning off linear interpolation approximately doubles the performance of the SRC at the expense of decreasing the quality of the final output.
- FIR Filter By setting ptsPerLobe equal to 1, turning lerp off and ratio equal to unity, the SRC can be turned into an nth order FIR filter (n equals the value stored in numofLobes). The coefficients for the FIR filter are stored in the first nth locations of the RAM coefficient table.
- Gain Section The gain section can adjust the output signal level from 0 to 8 times its current level with a step size of 1/4096.
- Speed The SRC will be able to convert a minimum of eight 48kHz voices in real-time, with linear interpolation on and 8 points of interpolation of the input signal.

Software

To perform a sample rate conversion on a block of data, an input and output Channel Program (CP) must be created and their respective Channel Program Pointers (CPP) must be passed to Mazda.

The following defines a Channel Control Word (CCW) for the sample rate conversion section of Mazda:



bits	Flags
55	Ready (0=XJS;1=Mazda)
54	Data Chain
53	Update
52	Interrupt
51	Timer Int
50	Aborted

Commands	
Download Parameters	0
Read Data	1
Write Data	2
Read Status	3
Jump	255

figure 5.10 - SRC Channel Control Word Definition

- **Jump:** The jump command causes MAZDA to update its current CPP to the location specified by the address value and continue executing CCWs from this location.
- **Read Data:** This command specifies the location and quantity of data to be DMAed into the SRC.
- **Write Data:** This command specifies the location and quantity of data to be DMAed out of the SRC.
- **Read Status:** This command specifies the location that the resulting status of the sample rate conversion should be written to. This command should be used at the completion of a sample rate conversion and be followed by an interrupt.

  The following is the format of the status word:

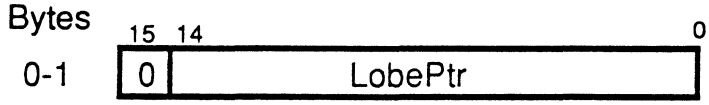


figure 5.11 - SRC Output Status Format

- **Download Parameters:** This command initializes the internal registers in the SRC to the values specified at the download parameter pointer. This command must be used before the start of a sample rate conversion and must be followed by a read data command.

  The following is the format and definition of the download parameters:

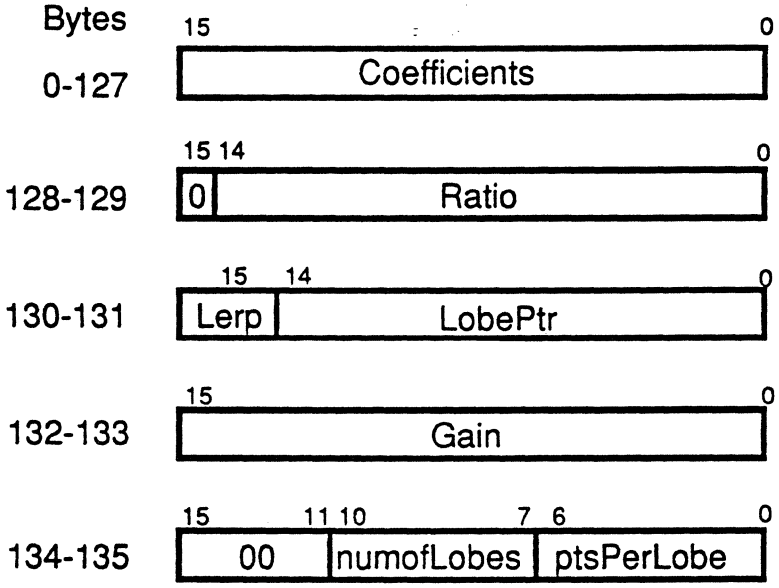


figure 5.12 - SRC Download Parameter Format

Sample Rate Converter Parameters Definition

- **Ratio**: sample rate conversion ratio (15-bits) =

$$\frac{\text{input sample rate} * \text{ptsPerLobe} << 8}{\text{output sample rate}}$$

- **numofLobes**: number of points of interpolation (4-bits). NumofLobes is equal to the number of multiplies between a sample and coefficient per output sample. (note: 0 = 16)

- **lobePtr**: current pointer into the RAM coefficient table (15-bits)

  7+8 bits; 1st 7 bits = sample offset + initial coefficient table offset; last 8 bits linear interpolation value. Initially set to

  ((((numofLobes/2)+1) * ptsPerLobe)<<8)

  This will allow the SRC to initialize the sample FIFO before interpolating the first output sample.

- **lerp**: linear interpolation of the coefficient table on or off (1-bit)

- **gain**: amplification level (16-Bits) : 1 sign bit, 3-bits of integer and 12-bits of fraction. For example;

Gain	Value
2	0x02000
1	0x01000
1/2	0x00800

- **ptsPerLobe**: points per lobe (7-bits)

  ptsPerLobe = 128/numofLobes

  for example;

numofLobes	ptsPerLobe
16	8
8	16
6	24
4	32
2	64

- **Coef** A 64 x 16-bits table of coefficients to be downloaded into the RAM coefficient table before starting the sample rate conversion. Note: only half of the coefficient table is downloaded into the sample rate converter since the table is assumed to be a mirror image.

The first step to initiate a sample rate conversion is for the XJS to create a CP for the output DMA controller (figure 5.13), and send Mazda a CPP to its definition. The output CP defines a data chain of the locations to place the output samples and the location to place the output status (figure 5.11).

63 Cmnd 56	55 Flags 48	47 Count 32	31 Address 0
Write Data	1 1 1 0 0 0	Data Count (1)	Write Data Pointer (1)
Write Data	1 1 1 0 0 0	Data Count (2)	Write Data Pointer (2)
	• • •		
Write Data	1 0 1 0 0 0	Data Count (n)	Write Data Pointer (n)
Read Status	1 0 1 1 0 0	2	Read Status Pointer (n)
Jump	1 0 0 0 0 0	Undefined	Top of CCW Pointer

figure 5.13 - SRC Output DMA Channel Program

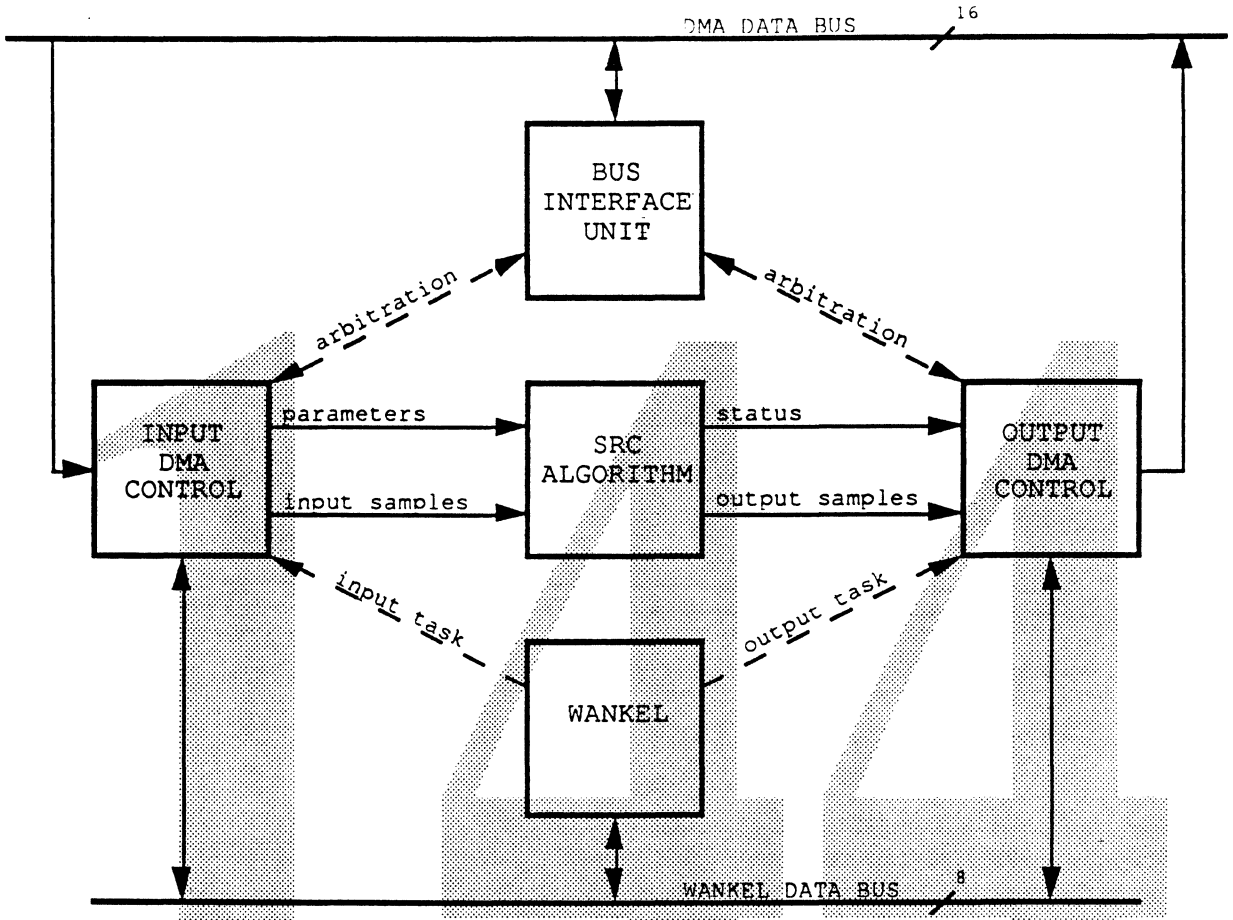
The second step is for the XJS to create a CP (figure 5.14) for the input DMA controller, and send Mazda a pointer to its location. The input CP includes a pointer to the location at which the download parameters are stored (figure 5.12) and a data chain of the locations to find the input samples . At this point, the SRC starts processing data and continues until it runs out of input samples, at which time Wankel will write out the status and interrupt the XJS.

63 Cmnd 56	55 Flags 48	47 Count 32	31 Address 0
Download	1 0 1 0 0 0	136	Download Parm Pointer
Read Data	1 1 1 0 0 0	Data Count (1)	Read Data Pointer (1)
Read Data	1 1 1 0 0 0	Data Count (2)	Read Data Pointer (2)
	• • •		
Read Data	1 0 1 0 0 0	Data Count (n)	Read Data Pointer (n)
Jump	1 0 0 0 0 0	Undefined	Top of CCW Pointer

figure 5.14 - SRC Input DMA Channel Program

Hardware

The Mazda I/O Controller chip contains a section for performing sample rate conversions. This hardware looks and acts like a pair of I/O Modules, one transferring a block of data (input samples) from memory to the sample rate conversion (SRC) hardware and the other transferring a block of data (output samples) from the SRC hardware to memory. Between these DMA controllers is a section of hardware that implements the SRC algorithm. The intent is to make sample rate conversion fit into the general method of transferring data on Mazda; driven by Channel Command Programs and Wankel tasks. The following figure is a block diagram of the sample rate conversion interfaces.



Here is the flow of control for performing a sample rate conversion:

1. XJS creates a Channel Command Program (CCP) for the output DMA controller and sends the pointer (CCPP) for this program to Mazda. The CCP specifies the location in memory for the output samples to be placed. Data chaining may be used if the destination area is not all contiguous. In addition, the output CCP contains a Channel Command Word (CCW) for reading status at the conclusion of the transfer.

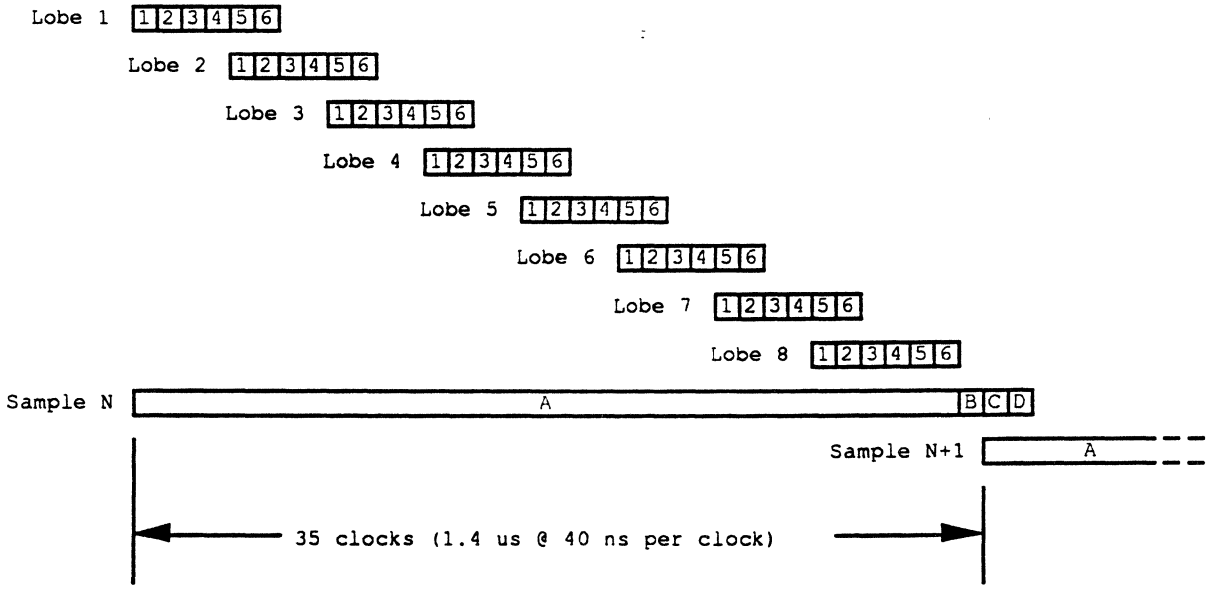
2. The Wankel task that controls the sample rate conversion output DMA controller starts executing the CCP created in step 1. However, since the input has not started yet, there is nothing to transfer and the DMA controller simply waits for data to start exiting the SRC hardware.

3. XJS creates a CCP for the input DMA controller and sends the pointer to Mazda. This CCP consists of a CCW for downloading the parameter block and one or more CCW's that specify the location in memory where the input samples reside. Once again data chaining may be used if the data is not contiguous.

4. The Wankel task that controls the input DMA controller starts executing the CCP created in step 3. The parameter block that is downloaded by the first CCW is the set of starting parameters for the sample rate conversion. After the parameter block has been downloaded, the flow of input samples starts. This in turn triggers the output DMA controller to start writing output samples to memory. The process continues until all the input samples have been used, at which time the output Wankel task stores the requested status, then interrupts XJS.

Timing

With Interpolation Enabled



The above figure is a timing diagram for producing an output sample with linear interpolation enabled. This operation is divided into two major steps. First an unamplified output sample is produced. On the timing diagram, this occurs during time A on the "Sample N" bar. The second major step is to multiply the unamplified output sample by a Gain value to produce the final result. This occurs during Cycles B, C, and D.

Step A is further divided into substeps; one for every lobe or point of interpolation. The diagram above assumes that the number of lobes is 8; this produces a total time per output sample of 35 clocks. The equation for determining the number of clocks is (4*numofLobes+3). The processing of each lobe's contribution to an output sample takes 6 cycles, but 2 of these cycles are overlapped with the previous lobe. The following is a description of the 6 processing steps per lobe:

Cycle 1: As the first step in interpolating the two values from the Coefficient Memory, COEF1 is subtracted from COEF2. This is a 16-bit subtract performed in the Upper Adder. The adder used in the SRC hardware is 26 bits wide, but can be divided and used separately as a 16-bit adder (the Upper Adder) and a 10-bit adder (the Lower Adder).

Cycle 2: The next step in interpolating the two values is to multiply the 8-bit field FRAC by the result of cycle 1. FRAC is one of the parameters downloaded at the start of the conversion. During this cycle, the multiplier performs the first part of this operation, creating the partial sum and partial carry for the adder to finish.

Cycle 3: The final interpolation step is to add the result of the multiplication to COEF1. Because the output of the multiplier is in partial sum and carry form, Upper Adder and Lower Adder must be 3-input adders.

Cycle 4: Now that we have an interpolated coefficient, the next step is to multiply this value by the sample data. This will require two passes through the multiplier to produce the final partial sum and carry results. On this cycle, the least-significant half of the multiply occurs.

Cycle 5: The second half of the multiply takes place in this cycle. The partial sum and carry from the Cycle 4 result are fed back into this operation, producing a final partial sum and carry for the entire multiply. Note that this cycle is overlapped with Cycle 1 of the next lobe sequence. This does not cause a resource conflict because Cycle 1 of the lobe sequence uses the Upper Adder and Cycle 5 uses the multiplier and Lower Adder.

Cycle 6: The final addition for this lobe takes place in this cycle. This calls for adding the partial sum and carry from Cycle 5 to the contents of the Sum Register (SUMR). SUMR accumulates the output sample value as each lobe adds its contribution to it. This cycle is overlapped with Cycle 2 of the next lobe sequence. Once again there is no conflict since Cycle 2 uses the multiplier and Cycle 6 uses the adder.

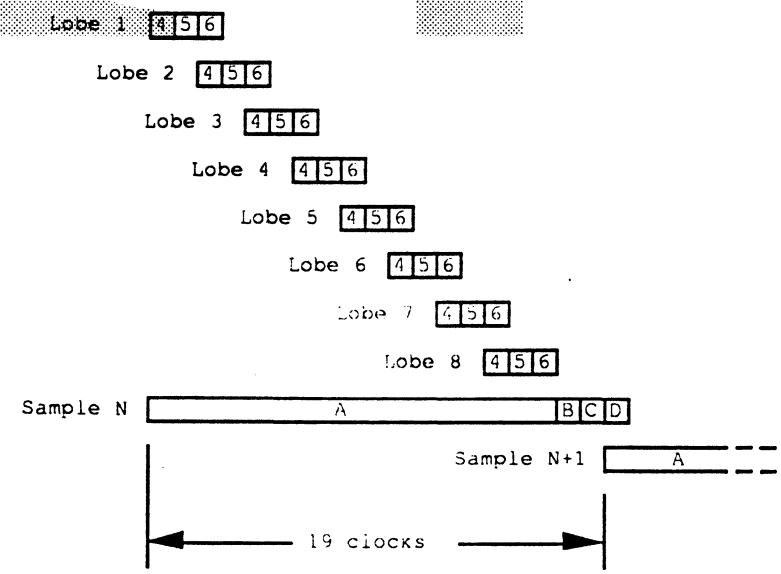
The Gain multiplication takes 3 cycles to complete (cycles B, C, D on the timing diagram). The following is a description of the processing steps that occur during these 3 cycles.

Cycle B: The Gain multiplication requires two passes through the multiplier. This cycle is used for pass 1, producing a partial sum and carry.

Cycle C: The second half of the Gain multiply takes place in this cycle. The partial sum and carry from the Cycle B result are fed back into this operation, producing a final partial sum and carry for the entire multiply. Note that this cycle is overlapped with Step A, Lobe 1, Cycle 1 of the processing for the next output sample.

Cycle D: The final addition for this output sample takes place in this cycle. The partial sum and carry from Cycle C are added together. If the result is an overflow (greater than a 16-bit 2's complement number), the output sample is forced to the maximum value that can be represented in 16 bits. This cycle is overlapped with Step A, Lobe 1, Cycle 2 of the next output sample.

With Interpolation Disabled



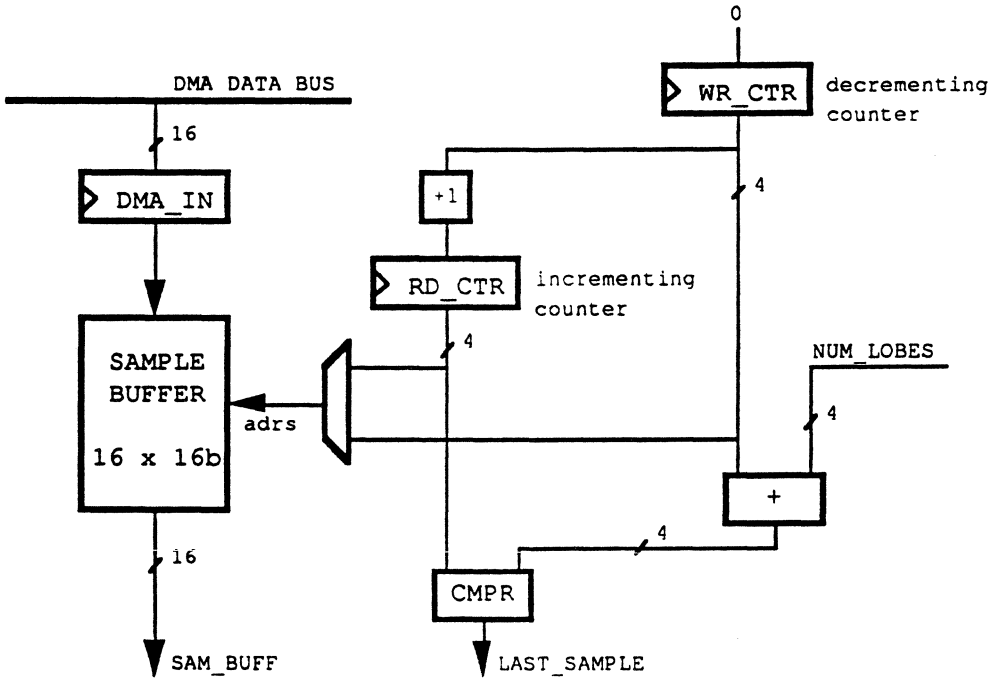
The above figure is a timing diagram for producing an output sample with linear interpolation disabled. Note that lobe processing requires only 3 cycles (with one overlap cycle) in this case. These cycles are

labeled 4-6 because they correspond to those steps on the "with interpolation enabled" timing diagram. The equation for determining the number of clocks per output sample with linear interpolation disabled is (2*numofLobes+3).

Hardware Architecture

Sample FIFO Buffer

The Sample FIFO Buffer is a 16-word by 16-bit circular queue used for holding the input samples. The following figure is a block diagram of this portion of the SRC hardware.



This hardware performs the following functions:

1. When the SRC algorithm is ready for a new input sample to be "shifted" in and if the next input sample has been acquired by the Input DMA Controller, the contents of the DATA_IN register are written to the Sample FIFO Buffer at the WR_CTR address. The WR_CTR is then decremented by one. This operation has the effect of shifting all entries of the FIFO forward one slot.

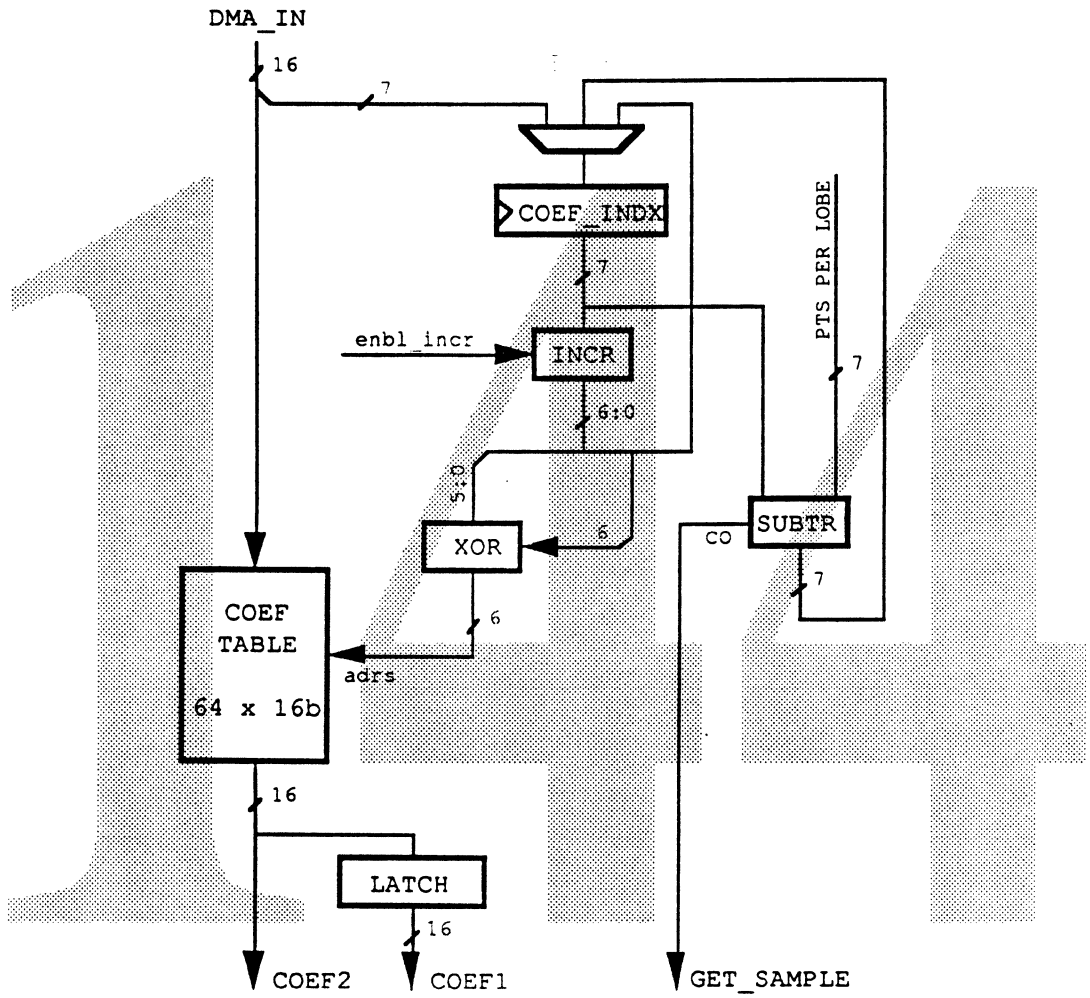
2. When the SRC algorithm is ready to process an output sample, RD_CTR is loaded with the value (WR_CTR + 1). This makes RD_CTR point to the latest entry in the FIFO. When this entry has been processed by the SRC algorithm, RD_CTR is incremented to point to the next entry. This process continues until the value in RD_CTR equals the modulo 16 addition of WR_CTR and NUM_LOBES (number of lobes). When this occurs, a signal is sent to the SRC algorithm indicating that the current FIFO entry is the final one to be processed.

Coefficient Table

The Coefficient Table is a 64-entry by 16-bit memory that holds the coefficient values to be used for an entire sample rate conversion. The "virtual" size of the table is actually 128 entries, but the second half is a

"mirror image" of the first half. Thus the contents of an address within the non-existent second half can be found by taking the 1's complement of the address and applying it to the first half.

The following figure is a block diagram of this portion of the SRC hardware.



The Coefficient Table is loaded through the DMA channel. This takes place during the "setup" Channel Command Word that downloads all the parameters for the sample rate conversion. The COEF_INDX register is initialized to zero, and it increments through the addresses as data is received in DMA_IN.

Before starting the SRC algorithm, COEF_INDX is initialized by the download CCW to a starting value of (((NUM_LOBES/2)+1)*PTS_PER_LOBE). This will cause the SRC algorithm to initialize the Sample Buffer before starting to produce any output samples. Here is the sequence of operations performed during the SRC algorithm by the hardware in the figure above:

1. PTS_PER_LOBE is subtracted from COEF_INDX. If the result of this operation is positive (CarryOut=1), the result of the subtraction is loaded back into COEF_INDX and a new input sample is shifted into the Sample Buffer (see the earlier description of operation for the Sample FIFO Buffer). Then

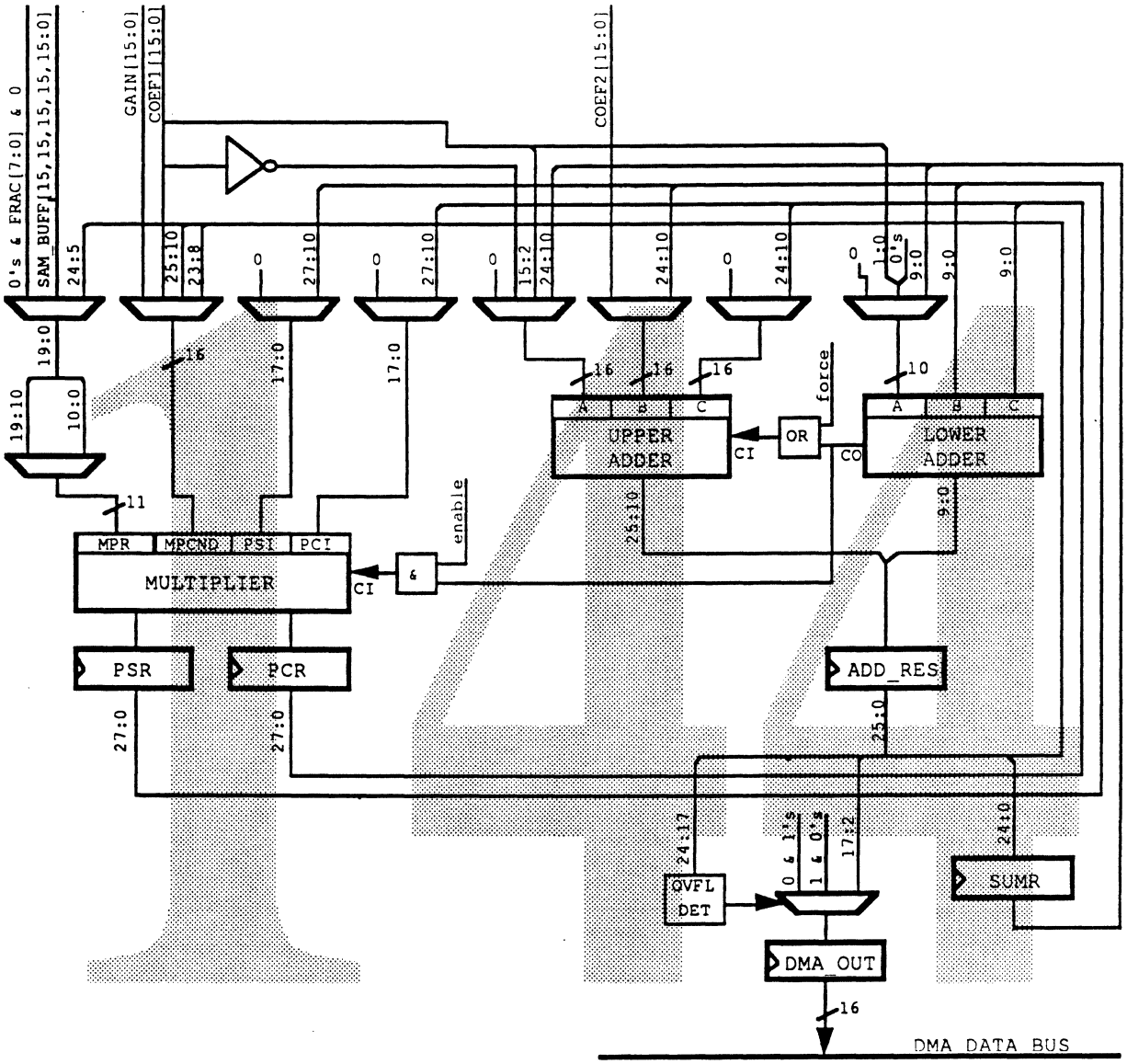
this step is repeated. If the result of the subtraction is negative (CarryOut=0), COEF_INDX is not changed and the process of producing an output sample begins with step 2.

2. The Coefficient Table entry at the address indicated by COEF_INDX is loaded into a latch (COEF1 output). If the most-significant bit of COEF_INDX is set, the address is negated before applying it to COEF_TABLE. If linear interpolation is disabled (LERP=0), this concludes the operation of the Coefficient Table hardware for this output sample. If linear interpolation is enabled (LERP=1), operation proceeds to step 3.

3. The address being applied to COEF_TABLE is incremented by one. Once again the 1's complement operation occurs if the MSB of this address is a one. The address to COEF_TABLE is held at this value during the interpolation that follows. COEF1 and COEF2 are the data values that are interpolated.

SRC Computation Hardware

The SRC computation hardware consists of the Multiplier, the Lower Adder, the Upper Adder, several multiplexers for selecting inputs to these computational units, and several registers for holding computational results. The following figure is a block diagram of this portion of the SRC hardware.



The Multiplier is capable of performing a 10-bit by 16-bit multiplication in one cycle or a 20-bit by 16-bit multiplication in two cycles. The output of the Multiplier is not a complete product; it is only the partial sum and partial carry for the multiplication. A full adder must be used to produce the final product. For a two-cycle multiplication, the Multiplier has the provision for circulating the partial sum and carry from the first cycle back into the Wallace Tree for the second cycle.

The Lower Adder is a 3-input 10-bit adder. It is separated from the Upper Adder for those cycles when the Upper Adder is being used for an addition and the Lower Adder is being used to generate a carry into the Multiplier.

The Upper Adder is a 3-input 16-bit adder. It may be used by itself as a 16-bit adder or it may be combined with the Lower Adder to form a 26-bit adder.

The Overflow Detection logic checks to see if the final output sample exceeds a 16-bit representation. If so, the output is forced to the maximum possible 16-bit positive or negative number, depending on the sign.

Software Simulation

The following is a "C" source listing for the sample rate conversion algorithm to be implemented in MAZDA.

```
/*********************************************************************
      Sample Rate Convert Hardware Simulation
*********************************************************************/

#define BUF_SIZE 16
#define UNDERFLOW -32768.0
#define OVERFLOW 32767.0

/************************************************/

extern short *inputPtr, *outputPtr;

short GetSample()
{
short a;
      a = *inputPtr;
      inputPtr++;
      return (a);
}

PutSample (sample)
short sample;
{
      *outputPtr = sample;
      outputPtr++;
}

/************************************************/

/*********************************************************************
      Sample Rate Convert a set of data
      •     input:      Ratio => sample rate convertion ratio
                              ((input sample rate << 8) * ptsPerLobe) / (ouput sample rate)
                        numofLobes => number of points of interpolation
                        coefPtr => pointer to the table of coefficents
                        lobePtr => current pointer into the coefficent table
                              7.8 bits;   lst 7 bits = sample offset + initial coefficent table offset
                                          last 8 bits linear interpolation value
                        numofSamps => number of input samples to interpolate
                        lerp => linear interpolation on/off
                        gain => amplification level 4.12
                        ptsPerLobe => points per lobe
                                    numofLobes:       ptsPerLobe:
                                    8                       16
                                    6                       24
                                    4                       32
                                    2                       64

      •     output:     lobePtr => current lobeptr
*********************************************************************/
unsigned short SRC (ratio, numofLobes, lobePtr, coefPtr, numofSamps, lerp, gain, ptsPerLobe)
unsigned char numofLobes, lerp, ptsPerLobe;
unsigned short gain, ratio, *coefPtr;
long numofSamps, lobePtr;
{
      register int i, coefIndex, j;
      register long sum, frac;
      short y1, y2, coef;
      short buf[BUF_SIZE];
```

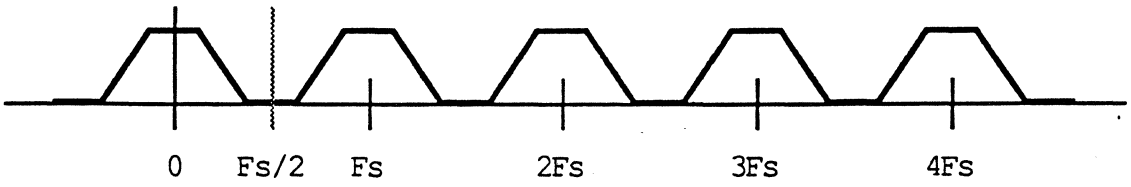
```
Open_Converting("\pSample Rate Converting...");

for (i=0;i<BUF_SIZE;i++) buf[i] = 0;

while (numofSamps--)
{
      while (lobePtr < ptsPerLobe<<8)
      {
            coefIndex = lobePtr>>8;
            frac = lobePtr & 0xFF;
            sum = 0;
            for (i=0; i<numofLobes; i++)   /* generate the coefficents */
            {
                  if (coefIndex & 0x40)    j = (coefIndex ^ 0x3F) & 0x3F;
                  else  j = coefIndex;
                  y1 = coefPtr[j];

                  if ((coefIndex+1) & 0x40)      j = ((coefIndex+1) ^ 0x3F) & 0x3F;
                  else  j = (coefIndex+1);
                  y2 = coefPtr[j];

                  if (lerp)          /* if linear interpolation is on */
                        coef = y1 + ((frac * (y2 - y1))>>8);   /*linear interpolate sinc tab */
                  else               /* if linear interpolation is off */
                        coef = y1;

                  sum += (long)coef * (long)buf[i];
                  coefIndex += ptsPerLobe;
            }

            sum = sum>>15;

            sum = (sum * gain) >> 12;      /* gain section */

            if (((sum & 0xf8000) != 0) && ((sum & 0xf8000) != 0xf8000)) /*is the data clipped?*/
            {
                  if(sum & 0x80000)
                        sum = UNDERFLOW;
                  else
                        sum = OVERFLOW;
            }

            PutSample ((short)sum);        /* store the computed sample */
            lobePtr += ratio;
      }

      lobePtr -= ptsPerLobe<<8;
      for (i=numofLobes-1; i>0; i--)       /* shift the sample in the FIFO */
            buf[i] = buf[i-1];
      buf[0] = GetSample();                /* get the next sample */
      if ((numofSamps & 0x07f) == 0)
            Change_Converting(numofSamps);
}
Close_Converting();
return (lobePtr);
}
```

A Note on Sample Rate Conversion

How interpolation (upsampling) is performed

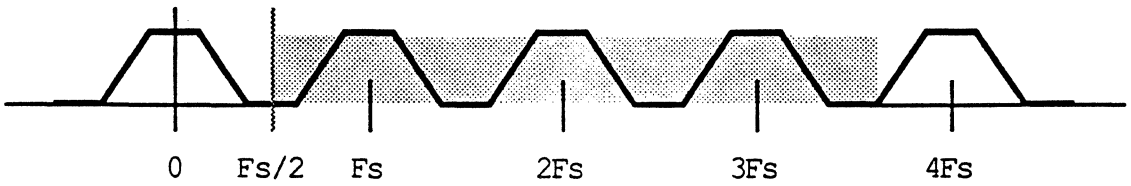
Suppose we want to translate a voice signal digitized at Fs = 8Khz to an equivalent spectrum sampled at 4Fs = 32 KHz. The following diagram shows the signal spectra to be interpolated. Since the signal was sampled in the digital domain, the spectrum is actually periodic at the original sample rate Fs = 8 KHz. The actual

sampled voice spectrum must be (according to Nyquist) band-limited to < Fs/2 = 4 KHz. The actual spectra we're interested in, then, extends from 0 to Fs/2.



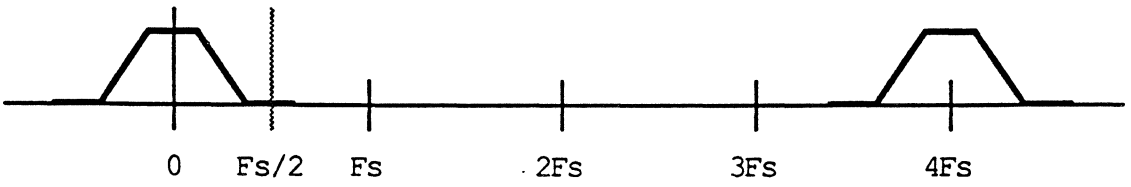
input signal spectrum to be interpolated

In order to create the equivalent spectrum sampled at 4Fs = 32 KHz, we must remove the aliased components at Fs, 2Fs, and 3Fs, as shown below. This is performed by a digital lowpass filter.



Shaded area must be filtering for 4x interpolation

Once this input spectra has been filtered, the result is the original voice spectra, but this time represented as a sequence of samples at the 4Fs = 32 KHz rate:

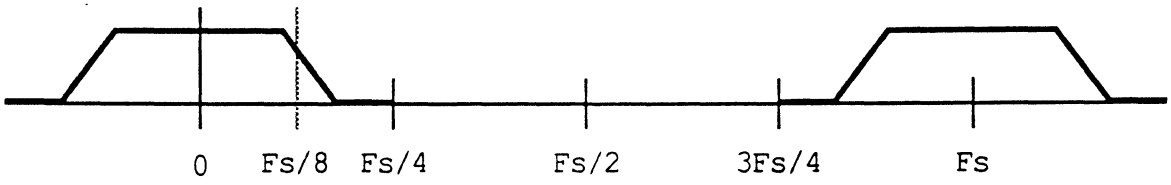


after filtering and interpolation,
resulting spectrum is periodic at 4Fs

Again, this signal is periodic around the new sample rate 4Fs. Note that at the new sample rate of 4Fs = 32 KHz, we could faithfully encode signals with frequency components up to 2Fs = 16 KHz. The extra samples are effectively redundant; no extra fidelity is obtained.

How decimation (downsampling) is performed

Suppose we have a microphone fed into a sampling system operating at Fs = 32 KHz. The spectrum looks like this:

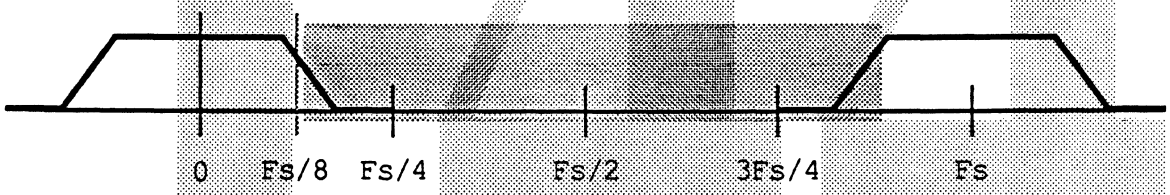


Spectrum to be decimated (downsampled)

We wish to convert this signal into a sample stream to feed a telephone CODEC at the standard 8 KHz = Fs/4 rate. Thus we have to somehow discard three out of every four original samples.

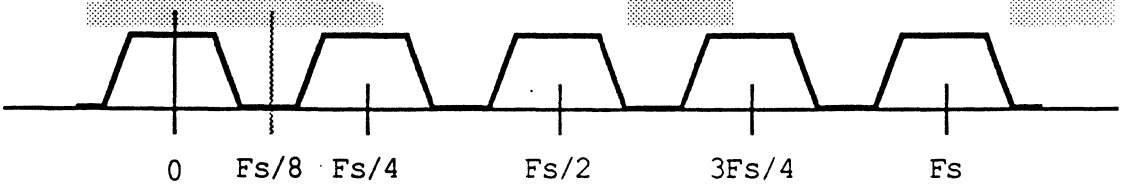
However, note that this signal contains frequency components beyond the Nyquist limit of Fs/8 = 4 KHz. If we simply took every fourth sample at 32 KHz and discarded the rest, the signal components beyond 4 KHz would be folded back or *aliased* into the original spectrum. The resulting spectrum would then be permanently distorted.

Instead, we perform a lowpass filtering operation analogous to that done in the interpolation case. The lowpass filtering operation is performed at the original sample rate Fs; then every fourth sample is taken from the filter output to create the desired 8 KHz sample stream.



Shaded area must be removed prior to decimation to avoid aliasing

The output of the lowpass filter has a spectrum resembling one taken using an Fs/8 = 4 KHz sampling rate:



spectrum after decimation to Fs/4

Note that, as a result of the decimation process, frequency components above Fs/8 = 4 KHz have been removed. Also note that, if the original signal had been bandlimited to < Fs/8 HZ before being sampled at Fs, no lowpass filtering would be required by the decimator. We could simply take every fourth sample from the input sample stream.

Section 7

Miscellaneous Interfaces

ROM

One of the interfaces to Mazda is the system ROM. This ROM contains the code for Wankels's task0 and the initial boot code for XJS. For a description of the initialization procedure, refer to the subsection on task0 in the Wankel section.

The ROM is one of the devices that sits on the low-speed bus. It uses all 8 data lines and all 16 address lines of the low-speed bus. The only other signal required is the ROM's chip enable line.

Transfers from the ROM are controlled by Wankel's DMA Controller. The destination for ROM data may be either the Wankel Code RAM (when downloading Wankel's task0 code) or system memory (when downloading XJS boot code).

CLUT

Mazda is responsible for controlling the MPU interface to the AC842 CLUT/DAC chip. The MPU port of the AC842 allows direct access to the internal control registers and color lookup table. The dual-port lookup table RAM allows color updating without contention with the display refresh process. For more details on the operation of the CLUT/DAC chip, refer to the AC842 specification.

The following signals are required by the MPU port of the CLUT/DAC chip:

D[7:0]	8-bit bidirectional data bus
A[1:0]	2-bit address bus (input to CLUT/DAC)
R/W	read/write signal(input to CLUT/DAC)
CS*	chip select signal(input to CLUT/DAC)

The MPU port of the CLUT/DAC chip is connected to Mazda's low-speed bus. It uses all 8 data lines of the low-speed bus. The 2-bit address port is connected to bits 1:0 of the address bus. The R/W signal is connected to bit 2 of the address bus. Only the CS* signal is a direct connection from Mazda.

The following is a list of Channel Commands to be executed by the Wankel task and I/O Module associated with the CLUT/DAC chip:

Write Address Register

Read Address Register

Write Pixel Bus Control Register

Read Pixel Bus Control Register

Write CLUT

Read CLUT

Write Test Registers

Read Test Registers

A typical Channel Program for loading the CLUT would consist of two Channel Command Words. The first CCW would be a "Write Address Register" with the data that is transferred to the chip being the starting CLUT address. The second CCW would be a "Write CLUT" with the Length field of the CCW determining the number of bytes that are transferred and therefore, the number of CLUT locations that are written.

Elmer

Mazda is responsible for controlling the MPU interface to the Elmer Frame Buffer Back End Controller. The MPU interface of Elmer allows read and write access to the chip's internal control and status registers. For more details on the operation of Elmer, refer to the Frame Buffer Back End ERS.

The following signals are required by the MPU port of Elmer:

D[7:0]	8-bit bidirectional data bus
A0	address line (input to Elmer)
R/W	read/write signal(input to Elmer)
CS*	chip select signal(input to Elmer)

The MPU port of Elmer is connected to Mazda's low-speed bus. It uses all 8 data lines of the low-speed bus. The 1-bit address port is connected to bit 0 of the address bus. The R/W signal is connected to bit 2 of the address bus. Only the CS* signal is a direct connection from Mazda.

When A0 is a 0, the operation (read or write) is to Elmer's internal 8-bit Index Register. When A0 is a 1, the operation is to the internal 8-bit register selected by the Index Register. After a read or write is

performed with A0=1, the Index Register is incremented. This facilitates block reads and writes of multiple registers.

The following is a list of Channel Commands to be executed by the Wankel task and I/O Module associated with Elmer:

Load Index Register

Write data

Read data

The low-level protocol for executing Elmer reads and writes is handled by the I/O Module. A write to one of Elmer's internal registers is performed by first executing a "Load Index Register" Channel Command to set up the address of the destination register, followed by a "Write Data" Channel Command. A read is performed similarly, except that the "Read Data" Channel Command is used. If the Count field of the Channel Command is greater than one, it indicates a block read/write. The I/O Module continues transferring data to/from Elmer (A0=1). The Index Register is auto-incremented by Elmer after each transfer.

Frame Buffer Vertical Line Counter

Mazda assists in the production of tear-free video updates. Refer to the Expansion Interface & Wilson ERS and the Frame Buffer Back End ERS for a description of the process of producing tear-free video updates. Mazda provides a 12-bit counter that closely tracks the value of the vertical line counter in the Back End logic. Since the XJS processor has no direct access to the Back End, the Vertical Line Counter in Mazda can be read by the processor whenever it needs to determine the current vertical line position of the Frame Buffer.

The counter in Mazda operates on a 40ns clock and is controlled by two signals from the Back End, VLC_CLEAR and VLC_COUNT. These two signals control the counter each 40ns cycle according to the following table:

VLC_CLEAR	VLC_COUNT	Action
0	0	no change
0	1	increment counter
1	X	clear counter

The counter exists as a read-only register in the processor's address space. XJS may read the value of the counter by issuing a word read. (See Mazda Memory Map for the address of this counter.)

System Timer

Wankel provides a high resolution 32 bit count-down timer for use by the operating system. It is clocked at a rate of approximately 1MHz (precise frequency TBD). This timer appears as a read/write register in the processor's address space. (See Mazda Memory Map for the address of this

timer.) The timer operates as follows: System software will write an initial count to the timer which will then begin counting down. When the count reaches zero an interrupt will be generated, but the count will continue to decrement. When the interrupt is serviced, the system can determine the exact amount of time which has expired since the timer was loaded by reading it's current value to see how far it has advanced past zero.

Wankel Timer

Wankel also contains a 32 bit free-running counter which is clocked at a rate of approximately 10 KHz (precise frequency TBD). This timer can be accessed both by XJS as a register in its address space (see Mazda Memory Map for the address), and by the internal Wankel processor. Wankel tasks can make use of this timer for a variety of functions. A particular application which is anticpated involves the MIDI interface. For inbound data, the Wankel task will use this timer to time-stamp the arrival time of each byte. On outbound transfers, the transmission time of the data can be controlled with a Channel Command which causes the channel to suspend until a time which is specified in the address field of the command has been reached. The access manager needs to be able to read this timer in order to determine the appropriate time value to write in the Channel Command.

It is also anticipated that this timer will be used to support a style of read operation in which an interrupt is generated and the Channel Command is updated in memory, if some number of characters have arrived within a given time interval. This will relieve the processor from being interrupted on the arrival of every character, but will still guarantee timely response to input data.

Apple Desktop Bus

The keyboard and mouse devices will be supported via ADB. This interface will be implemented in the ASIC which controls the other desk top devices.

[Note that two issues might affect this strategy: 1) If we use an Industrial Design which has one unified box on the desk top, the ADB interface would be implemented within Mazda. 2) If we decide to use ChefCat as our connection to the desktop, and there is a strong commitment to implement keyboard and mouse peripherals with ChefCat interfaces, there would be no need for ADB at all.]

[ADB channel program definitions to be added].

Real Time Clock

The system requires a battery backed up real time clock for maintaining time and date while the system is powered down. This chip should also provide a small amount of non-volatile RAM for storing certain parameters while powered down.

The Big Ben real time clock chip is one of the alternatives being considered for this function. Big Ben provides a 47 bit constantly running clock as well as 512 bytes of non-volatile parameter RAM.

The real time clock chip will be connected via the low-speed bus.

[RTC channel program objects and method definitions to be added].

System Controller Registers

Mazda will provide access to the status and control registers of the System Controller (SC) ASIC. The SC will be connected via the low-speed bus. It contains several 18 and 32 bit registers which must be programmed with multiple transfers on the 8 bit data bus. See the Memory System ERS for details regarding the SC registers.

[SC channel program objects and method definitions to be added].

Section 9

Hardware Implementation

Mazda Memory Map

Address Range	Register	R/W	Bit Width
0x84000000	CPP0	W	32
0x84000008	CPP1	W	32
:	:	W	32
0x840001F8	CPP63	W	32
0x84000200	IRQUpMask0	R/W	32
0x84000208	IRQLoMask0	R/W	32
0x84000210	IRQUpMask1	R/W	32
0x84000218	IRQLoMask1	R/W	32
0x84000220	IRQUpNew0	R	32
0x84000228	IRQLoNew0	R	32
0x84000230	IRQUpNew1	R	32
0x84000238	IRQLoNew1	R	32
0x84000240	IRQIn	R	32
0x84000248	VertLineCount	R	12
0x84000250	MemReady	W	1
0x84000258	BootXJS1	W	1
0x84000260	SysTimer	R/W	32
0x84000268	WnklTimer	R/W	32

Note: Since Mazda interfaces only to the upper half of the XJS Data Bus, all memory-mapped addresses to Mazda are multiples of 8 (even word addresses).

Mazda Pinout

Signal Name	Mnemonic	Pin Type	Number of Pins
	pXXX = Active High		
	n XXX = Active Low		
Bus Transfer Signals			
Data	pD63-pD32	Input/Output	32 pins
Address	pA31-pA0	Input/Output	32 pins

Transfer Attribute Signals

Transfer Size	pTSIZE1-0	Output	2 pins
Transfer Burst	nTBST	Output	1 pin
Read/Write	pR/nW	Input/Output	1 pin
Lock	pLK	Output	1 pin

Transfer Control Signals

Transfer Start	nTS	Output	1 pin
Transfer Acknowledge	nTA	Input	1 pin
Transfer Error Acknowledge	nTEA	Input	1 pin
Transfer Retry	nTRTRY	Input	1 pin
Address Acknowledge	nAACK	Input	1 pin
Mazda Slave Request	nMSREQ	Input	1 pin
Mazda Slave Acknowledge	nMSACK	Output	1 pin

Snoop Control Signals

Intent to Modify	nIM	Output	1 pin
Memory Cycle	nMC	Output	1 pin
Snoop Retry	nSRTRY	Input	1 pin
Address Retry	nARTRY	Input	1 pin
Global	nGBL	Output	1 pin

Arbitration Signals

Bus Request	nBR	Output	1 pin
Bus Grant	nBG	Input	1 pin
Address Bus Busy	nABB	Input/Output	1 pin
Data Bus Busy	nDBB	Input/Output	1 pin
Data Bus Grant	nDBG	Input	1 pin

Interrupts

Processor Interrupt	nINT0-1	Output	2 pins
Expansion Slot Interrupts	nESI1-3	Input	3 pins
Back End Interrupt	nBEI	Input	1 pin

Vertical Line Counter Control

Line Counter Clear	nVLC_CLR	Input	1 pin
Line Counter Count	nVLC_CNT	Input	1 pin

Desktop Bus

Transmit Data		Output	1 pin
Receive Data		Input	1 pin

Control		Input/Output	2 pins
Eternet Interface			
Data		Input/Output	8 pins
Control		Input/Output	11 pins
SCSI Interface			
Data Bus	pD7-pD0	Input/Output	8 pins
Address Bus	pA3-pA0	Output	4 pins
Chip Select	nSCSICS	Output	1 pin
I/O Read	nIOR	Output	1 pin
I/O Write	nIOW	Output	1 pin
Interrupt Request	nIRQ	Input	1 pin
Data Request	pDRQ	Output	1 pin
Data Acknowledge	nDACK	Input	1 pin
Reset	nRST	Output	1 pin
Spock Interface			
Address	pA(0-2)	Output	3 pins
Read/Write	pR/nW	Output	1 pin
Data	pD(0-15)	Input/Output	16 pins
Chip Select	nDEV	Output	1 pin
Reset	nReset	Output	1 pin
Interrupt	nINT	Input	1 pin
DMA Request	nDRQ	Input	1 pin
DMA Acknowledge	nDACK	Output	1 pin
Data Strobe	nDS	Output	1 pin
Serial read/write Clock	CRYS	Input	1 pin
Analog Phone Interface			
Transmit Data		Output	1 pin
Receive Data		Input	1 pin
Off-hook			1 pin
Ring			1 pin
Low-speed Bus Interface			
Address		Output	16 pins
Data		Input/Output	8 pins
CLUT Interface			
CLUT Select	nCLUT_CS	Output	1 pin

Elmer Interface			
Elmer Select	nElmer_CS	Output	1 pin
ROM Interface			
ROM Output Enable	nROMOE	Output	1 pin
System Controller (SC) Interface			
SC Select	nSC_CS	Output	1 pin
ISDN Interface			
IDC Chip Select	IDCCS	Output	1 pin
IDC Write	IDCWR	Output	1 pin
IDC Read	IDCRD	Output	1 pin
IDC Interrupt	IDCINT	Input	1 pin
ISDN/SCC Mux Interface			
ISDN/SCC Mux Control		Output	3 pins
Serial Bus In	SBIN	Output	1 pin
Serial Bus Out	SBOUT	Input	1 pin
Serial Bus Frame Sync	SFS	Input	1 pin
Serial Clock	SCLK	Input	1 pin
B1 Channel Input	B1IN	Input	1 pin
B1 Channel Output	B1OUT	Output	1 pin
B1 Channel Tx/Rx Clock	B1CLK	Output	1 pin
B2 Channel Input	B2IN	Input	1 pin
B2 Channel Output	B2OUT	Output	1 pin
B2 Channel Tx/Rx Clock	B2CLK	Output	1 pin
SCC1 Interface			
Chip Enable	nCE	Output	1 pin
Interrupt	nInt	Input	1 pin
DMA Request A	nReqA	Input	1 pin
DMA Request B	nReqB	Input	1Pin
SCC2 Interface			
Chip Enable	nCE	Output	1 pin
Interrupt	nInt	Input	1 pin
DMA Request A	nReqA	Input	1 pin
DMA Request B	nReqB	Input	1Pin

Miscellaneous

System Reset	nRST_IN	Input	1 pin
Processor Reset	nRST_XJS0-1	Output	2 pins
Clock	pCLK	Input	1 pin
JTAG Pins		Input	5 pins
Power, GND Pins			50 pins
TOTAL			**279 pins**

Mazda Summary

Module	#Pins	#Gates	Bandwidth (MB/S)	Power (mWatts)	#Channel Programs	#Data Buffers	#Wankel Tasks
Wankel		5000		241			
Code RAM (4Kx16)		35000		1688			
Scratchpad RAM (256x8)		2128		103			
Context File (32x7)		560		27			
PC File (128x16)		4685		226			
Channel Program Central		2000		96			
Channel Prog. RAM (64x96)		4608		222			
Data Buffer RAM (Bufs*8x32)		9830		474			
XJS Bus Interface		5000		62			
Data	32			114			
Address	32			19			
Ctrl	21			12			
Sample Rate Conversion		2000	2.00	96	2	1	1
Coefficient RAM 64x16		1229		59			
System Timer		1000		48	1	1	1
Desktop Bus		3000	2.00	145	8	8	8
Data	2						
Ctrl	2						
Ethernet Interface		1000	1.25	48	2	2	1
Data	8			14			
Ctrl	11			19			
SCSI Interface		1000	3.00	48	8	1	1
Data	8			33			
Ctrl	11			45			
Spock Interface		1000	2.00	48	1	1	1
Data	16			22			
Ctrl	10			14			
Analog Phone Interface		1000	0.02	48	2	2	2
Ctrl/Data	6						
Low Speed Bus		200					
Address	16						
Data	8						
Common Ctrl	2						
CLUT Interface		500	?		1	1	1
Ctrl	2						
ROM Interface		1000	?		1	1	1
Ctrl	1						
Memory Control Interface		300	?		1	1	1
Ctrl	1						
ISDN Interface		1000	0.004	48	2	2	2
Ctrl	3			0			
2B+D Mux	8		0.02	0	2	2	2
SCC Interface 1		1000	0.02	48	4	4	4
Ctrl	7			0			
SCC Interface 2		1000	0.04	48	4	4	4
Ctrl	7			0			
Clock Chip (Big Ben)		300	0.00	0	1	1	1
Ctrl	4			0			
Total Wankel Tasks							31
Total Data Buffers						32	
Total Channel Programs					40		
Total Power				4120			
Total I/O Bandwidth			10.34				
Memory Real Estate (gate equiv)		58040					
Logic Real Estate		27300					
Total Gates		85340					
Interrupt Pins	6						
JTAG Pins	5						
Power, Gnd Pins	49						
Total Pins	278						

Desktop Connection (CLT)

The purpose of the desktop connection is to provide an interface between Mazda and the I/O devices that exist in the desktop box. The status of this interface is somewhat uncertain at this time, until a decision is reached on the existence of the desktop box. The description that follows assumes that there is a desktop box separate from the main CPU box.

As it pertains to the I/O subsystem, the desktop box is composed of the following I/O interfaces:

- Microphone jack and 48Khz stereo ADC
- Internal speakers, jack for external speakers, and 48Khz stereo DAC
- Handset jack and 8Khz CODEC
- Apple Desktop Bus (ADB) connection

In addition, the desktop box contains an ASIC that multiplexes/demultiplexes between the single data stream coming from Mazda and the multiple data streams of the desktop I/O devices. The desktop connection, i.e., the interface between Mazda and the desktop ASIC, could be either an implementation of the P1394 serial bus (ChefCat) or a design of our own specification. Using ChefCat would allow us to leave open the possibility of connecting removable storage devices on the desktop. On the other hand, rolling our own interface, one that is designed only to meet the requirements of the I/O devices listed above, is probably the approach with the least schedule risk. The Issues section at the conclusion of this ERS contains more on the pros and cons of ChefCat.

Low-speed Bus

The general concept in Mazda's connection to external peripherals is one in which a number of "I/O Modules" each control a dedicated port to a peripheral controller. This provides the maximum performance, and simplifies the I/O Modules. However, due to pin limitations, this approach can not be used for all of the I/O interfaces. Therefore, a number of the external devices will be attached to a common address and data bus. These devices all have relatively low bandwidth requirements, and they have fairly similar interface requirements. The additional latency introduced by the need to arbitrate for this bus will not present a problem for these devices.

The following devices will be attached to this bus: The SCC used for LocalTalk and the generic serial interface; the ISDN controller; the SCC used for ISDN's two B channels; the real time clock chip; the CLUT/DAC; the system ROM; and the System Controller ASIC. The connection to the System Controller is used to access its status and control registers.

In addition to the connection to this bus, each device will receive a dedicated chip-select signal, as well as a number of dedicated control signals, such as DMA request, and acknowledge.

The following signals are shared by the devices attached to the low-speed bus:

Data[7:0] All of the devices use this 8 bit bidirectional data bus.

Address[15:0] 16 bits of address are provided to allow connection to a large ROM. The real time clock chip also uses 10 bits to address its parameter RAM. Most other devices need four or less bits of address. Additionally, control lines which are conditioned with a chip-select to the device (e.g. Read, Write) will be connected to Mazda via unused address pins.

Section 9

Issues

Mazda

- Is this architecture really what the software folks want? An early presentation of this at the Pink Offsite suggests that they seem to like it. We await feedback to make sure that we are proceeding in the correct direction.

- Perhaps one good way of testing the viability of this whole architecture is to implement it on top of Opus. I.e., try to architect a version of Pink which mapped its I/O into the proposed Jaguar architecture and see what and how many problems are encountered.

- A closely related issue is: Does this architecture provide enough flexibility for third parties, etc.? The current Mac, for all of its kludginess, does provide users (developers) the ability to alter and/or extend the I/O system by directly interacting with the individual I/O Chips. A system built around an independent I/O Controller chip isolates the I/O system to a large extent. It is not clear how much of a negative reaction this will have upon the developer community. Of course, we could allow vendors to write I/O code for the chip. This makes the system more extensible, but does open it up to corruption, etc. by bad code, bugs, etc.

- How do we get "desktop" data between the main CPU box and the desktop if there are two separate boxes in a Jaguar implementation? What we are assuming is that a high-speed, serial communications path is provided. In some of the documentation, mention of "ChefCat" may appear. ChefCat is a serial bus being developed by PSAP. It does provide the necessary services; however, there is some question as to its reality. Regardless of the exact details, the current thinking is that a "ChefCat Like Thing" is needed to communicate the sound and ADB data between the user and the box.

- Is the Wankel code RAM large enough to hold all the task code? A related issue is whether we will need to dynamically alter the code RAM and scratchpad allocation when we add or remove tasks. A potential compromise position is that the majority of tasks are fixed, but that there are a few seldom used tasks that are swapped in and out of one or two fixed-size blocks in the code RAM.

- Should the ROM that holds the Wankel task0 code and the initial processor boot code actually be EEPROM? The disadvantage of EEPROM is that it is more expensive and it increases Mazda pin count and design complexity. The advantage of EEPROM is that it would ease development by eliminating the need to upgrade ROM. Also we could get surface mount chips that do not need to be hand stuffed, thereby improving reliability and perhaps overall cost.

Net/Telecom

Chefcat or equivalent bus: Placing codec/sound amplifier functions on monitor requires digital highway and associated controllers/protocol work. If the desktop connects directly to the telecom/sound subsystem, these design hurdles are removed. Plus, we can use the built-in codec in the AMD ISDN transceiver chip for the handset/headset. In any case, the handset-speakers-mikes must be in close proximity to the user.

Ethernet controllers: Rapid developments are underway in this area and more work is needed with vendors to verify chip performance, reliability and manufacturers' qualifications. Components under investigation provide functionality and interface similar to the Intel 82590 or SEEQ 8003 controllers.

CODECS: Suitable functionality is available "off shelf", but very promising codec designs have been recently proposed using sigma-delta technology. Wide price differentials amongst competing and promised parts must be investigated.

Real-time performance: ISDN controller, modem pseudo-devices, ethernet controller, sound synthesis: all have stringent real-time scheduling requirements. Maximum delay is:

- ISDN "B" channel: 2 ms at 64 Kbps with 32 byte FIFO. Higher level protocols can transcend error at controller level.
- modem pseudo-devices: around 20 ms with 160 byte DMA buffers. Controller level underruns will result in garbled transmissions.
- ethernet controller: about 38 µsecs with 48-byte FIFO. Higher level protocols can transcend error at controller level.

RALPH: The RALPH control program running on WANKEL.depends upon a quiescent (standby) power mode in order to detect and answer incoming phone calls with the power "off".

ISDN: The low-level datalink control program running on WANKEL.also depends upon a quiescent (standby) power mode in order to receive incoming calls with the power "off". The ISDN call management timeouts must be studied to identify constraints.

Mass Storage

- Maintenance of real time thread integrity between host and storage.
- Synchronization of multiple stored threads.
- Minimization of latencies.
- Selection of page size and block size.
- Spooling real time threads.
- Scatter/gather optimizations.
- Stream interleaving during storage i/o.

References

Sound Manager, Steve Milne

Zilog SCC Data Book

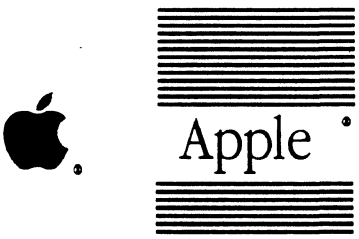
AMD ISDN data sheet for 79C30A, 79C32A, 1989

TI Telecommunications Component Data Book, 1989

Intel Microcommunications Data Handbook, 1989

Inside AppleTalk

144



Jaguar System Issues

External Reference Specification
Special Projects

Nov. 15, 1989

Return Comments to:
Ron Hochsprung
Phone: x4-2661
AppleLink: HOCHSPRUNG1
MS: 60-AA

144

Apple CONFIDENTIAL

Contents

Introduction

This chapter deals with several issues which impact the system as a whole, cutting across many different functional blocks of the design. While individual chapters may discuss their part of the whole, this chapter ties them together.

System Startup (Booting)

Note that in the following discussion we will use the generic term "ROM" to denote a class of devices which are considered to be "Read Only". In Jag1, the actual devices may be EEROM, which allows the "ROM" to be altered after being installed in the system. This would allow field upgrades of ROM without replacement of parts. However, since there is a limited amount of ROM on Jag1, this feature is somewhat moot.

The current Macintosh systems are implemented with ROM being an integral part of the run-time environment. Most of the system code (and some of its data) is executed directly out of this ROM. The width of the ROM is commensurate with the data path size of the CPU. In many Macs, the ROM is at least as fast as DRAM; in some, it is (effectively) faster. The ROM is seen as an important resource to the dynamics of the system.

Jaguar diverges from this "traditional" path. In a Jaguar, there is only a minimal amount of ROM. The ROM is considered as a static resource which is only important during the initial phases of system startup. During the booting process, an initial system image is copied from ROM (by Mazda) into DRAM, before the CPUs are taken out of reset. When the CPU(s) startup, they execute out of DRAM; the CPUs do not have direct access to the ROM.

There are several reasons for this model. The biggest reason is that it is expensive to support ROM in the same manner as in the Mac; the data path chips do not have the pins to accommodate wide ROMS. A secondary reason is that the ROMs are not as fast as the DRAM system (running in burst mode). Thus, the standard ROM approach would be more costly and slower!

Since every Jaguar has a guaranteed mass storage device, the system image (beyond that which is copied into DRAM from ROM) is kept on the on-board hard disc. The initial system image contains only enough code to perform a first-stage system checkout and initialization. After this first stage boot, the rest of the run-time system is loaded from the system disc area.

The process of "booting" occurs in the following steps:

1) At power-up (or, when the RESET button is pressed), all chips are reset.; the XJSs remain in the reset state.

2) Mazda exits the reset state by executing a hardware state machine which down-loads a section of the System ROM into Wankel's code memory.

3) After this initial code is loaded, Wankel begins execution. This first-stage boot code contains the initializations necessary to allow Mazda and the rest of the system to be

used by XJS code.

4) Wankel then copies a second-stage boot image into DRAM; this image consists of XJS code.

5) Wankel removes the XJS RESET signals, allowing the XJS to begin execution. Mazda/Wankel then enters a minimal I/O support mode.

6) The XJS begins execution. This stage of the boot code needs to ascertain the size and type of DRAM and initialize the System Controller with the appropriate control values. It also needs to perform secondary initialization of Mazda (e.g., down-loading the appropriate Spock image). The XJS then uses Mazda to read in the third-stage boot image from the "default" boot device (typically, the hard disc, but could be network or floppy).

7) This third-stage boot image could contain an entire "runable" image of the OS; more likely, it would correspond to the code which is currently being loaded by the Mac Pink startup program. At this point, the system is as functional as the current Mac is by having a larger, more complex ROM.

The above works for a single processor system. A minor perturbation of this handles a dual-processor system. In that case, the "first" XJS works as above. During that first stages, the "second" XJS is held in reset. The Reset Vector of the DRAM image is set by XJS0 to point to code which properly initializes XJS1 (e.g., loading one of its system registers with its "id"). XJS0 then "tells" Mazda to take XJS1 out of reset. Upon exiting reset, XJS1 will now execute its processor unique code before beginning its "normal" processing.

Obviously, this scheme can be extended to our 16 processor model... (The somewhat more expensive system).

Jag1 Coherency Model

The XJSs provide a cache coherency mechanism which is meant to guarantee that only one copy of modified data exists within caches and/or memory in a multi-processor XJS system. This mechanism works by providing "snooping" logic within the caches of the XJSs such that any request for (modified) data by one XJS which is currently within the cache of a second XJS will be written to memory before the first access is allowed to complete. For this to work, all memory accesses must be "seen" by all XJSs.

The Jag1 supports this XJS coherency model by tying all (memory) bus masters onto a single Address bus, as required by the XJS protocol. Thus, any access to Jag1 memory (either Main Memory or the Frame Buffer) will be potentially coherently managed. I say potentially, because the operating system must identify those areas of memory (via the Global bit in page table entries) which are to participate in the coherency protocol. Areas which do not require this mechanism (e.g., they are guaranteed unique, or you don't care!) can be marked as non-Global; this may allow some performance enhancements within the System Controller for access to such areas.

Cards on BLT do not participate in this coherency model when they transfer data amongst

themselves. That is, the BLT interconnect itself does not participate in this coherency. However, all Jag1 memory accesses from the BLT cards will be accessed using this coherency protocol.

To sum up, any memory area marked Global by the XJSs and all accesses by E&W (for both Wilson channels and BLT slots) participate in coherency. Once data is transferred to BLT, coherency is not guaranteed!

Interrupt System

Almost every functional block within the Jaguar has associated with it at least one interrupt. All of these sources of interrupts must be coordinated and centralized. This section presents a possible interrupt strategy for Jaguar. (Note: the interrupt model here refers to sources of the XJS INT* pin.)

The interrupt model of a multi-processor Jaguar implementation assumes that both CPUs are symmetric and homogeneous. That is, we assume that there is no "master" CPU to which interrupts must be directed. An argument as to why this should be the case is that it minimized the interrupt latency, since whichever processor can be interrupted first (i.e., has interrupts enabled) will process the interrupts. Jaguar 's interrupt system has been designed to be efficient in the context of this "equality" between the CPUs.

Within the Jaguar, the Mazda chip contains this centralized interrupt logic. The service provided by Mazda is implemented entirely by hardware. (Some of the interrupts may be generated as the result of Wankel code.) In order to allow a consistent environment for a multi-processor implementation, the interrupt system provides an "atomic" interrupt servicing paradigm.

Mazda implements two registers which reflect interrupt requests. One of these registers contains a bit for each interrupt source and dynamically reflects the state of the interrupt. This register can be "polled" at any time; bits which change only when the corresponding interrupt source changes state. For example, if a BLT slot requests an interrupt, that slot's bit in this register will continue to look set until the software handler for the slot performs some action which will cause the card to remove its interrupt request. At that point, the bit within the Mazda register will appear clear.

The second register, which is used to actually generate an interrupt to the CPU(s), is an "edge-sensitive" version of the interrupt sources. I.e., a bit will set in the second register only when the corresponding bit changes from a 0 to a 1 in the first register. This second register is the one which is intended to be read by the interrupt handler. Reading this register will clear all of its bits; the bits will remain clear until interrupt sources present an "edge" again.

In the BLT slot example, after a CPU reads the second register, that slot's bit will be cleared. It will remain clear until the slot removes the interrupt and asserts it again. This new assertion will cause an "edge" which will then be captured in the second register. This edge-sensitivity prevents continuous interrupts from devices which are managed by routines which do not immediately handle the source of interrupt (i.e., at interrupt service time). This "first time only" mechanism should allow Interrupt Service Routines (ISRs) to be more cleanly integrated into the operating system, since their immediate action is not required.

Note that if both CPU(s) are interrupted in a multi-processor system, whichever processor reads the second register "first" will see all of the current interrupts. Assuming that no intervening interrupt gets signalled between the reads of the first and second CPU, the normal case, the second CPU will read all zeros from the

register. Upon reading zeros, the second CPU can immediately exit its interrupt handler. Only if an interrupt (edge) is detected between the two reads would the second CPU read a non-zero value.

Because of the edge sensitive nature of the interrupt system, some special handling may be necessary to make sure that interrupts are not dropped. For example, if a BLT card has several sources of interrupts which must be "multiplexed" onto the single interrupt request line, the Interrupt Service Routine (ISR) for that card must take special care to make sure that all of its potential sources of interrupt are handled. This simply means reading the active interrupt state register to determine if its processing has indeed removed the interrupt request. If not, it must continue processing until the line becomes clear.

In order to support some types of "ASAP" interrupts (e.g., a BLT slot-to-slot interrupt with minimal latency), a special mechanism has been provided within Mazda which allows a channel program to be "triggered" by the occurrence of an interrupt. Using this feature, a BLT slot interrupt from one slot could cause a write to a "magic" location on another. This magic location write could cause an interrupt of the processor on the second card.

One XJS can interrupt another by writing to a such a "magic" location within Mazda's address space. This would be used in circumstances where the "other" XJS must be alerted to some change of system state. For example, when XJS0 changes the state of a page table entry, it may be necessary to interrupt XJS1 so that it can flush its on-chip PATC to make sure that it uses the correct (updated) entry.

Non-Maskable Interrupts (NMI)

The XJS also has a Non-Maskable Interrupt pin which forces an interrupt unconditionally. This interrupt mechanism is provided primarily for debugging, since it potentially destroys state information which prevents a resumption of the interrupted activity.

The current plan is to treat this line as we have in the past on Macintosh. The line will be pulled up, with an optional switch to ground it. It will have no interface with any other components of the system.

Bus Errors

The Jag1 will be capable of generating bus error responses for Reads. The normal source of these would be Parity errors on Main Memory or BLT Errors. For Reads, the error response can be signalled during the corresponding bus cycle, and hence, synchronous with the bus activity.

Writes, however, cause a problem since the Write transaction has been completed as far as the processor is concerned. This is once consequence of using the aggressive "dump-and-run" strategy for Writes. Main Memory will not generate Parity errors for a Write, so that possible source of Bus Errors is eliminated.

However, BLT slots can still cause an Error on a Write which "completed" as far as the XJS is concerned. For example, a card could signal Error when the Write is initiated (to the card); or, the BLT could detect a time-out condition.

Besides the fact that the processor has long passed the point at which the Write may have occurred, another problem can be induced by BLT Time-Out Errors. The problem is that many subsequent transactions may have been stacked up in the data patch and/or E&W chips. A potential for a "grid lock" on bus traffic could exist.

In order to mitigate these problems, BLT will signal an Error on a Write by using an interrupt to Mazda, which ultimately translates to an interrupt to XJS. The XJS can query BLT to determine the address of the Error. Most likely, the software would have to "crash" that offending task, since there is no guaranteed way of determining how to recover.

In addition to reporting the Error, BLT will immediately generate Error responses to any subsequent accesses to that slot; this includes any other processors and/or BLT slots which may be accessing that card. This guarantees that buffers will never back up so far that nothing gets done.

The trade-off for such indeterminacy on Writes is the much higher system throughput provided by dump-and-runs.

QUESTION: Can any one think of why the above strategy is too simple? This model assumes that Bus Errors are "catastrophic". Is this a reasonable assumption?

Power Control

The Jaguar has several potential usages which require a system which can respond "immediately" to events. For example, if we want to be able to receive FAXes at any time, the Jaguar must be able to answer the phone and start receiving data in a timely manner. However, the user may not want to leave his/her system turned on all the time.

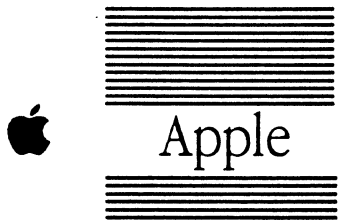
The current thinking is that when the user "turns off" the Jaguar, it does not entirely turn off. Rather, it enters a "reduced power" mode which give the illusion of being off (e.g., the disc and fan stop), but which is alive enough to provide "instant turn-on".

In this model, Mazda is able to control the reduced power mode by explicit power down circuitry (e.g., control of the fan?) and by using the RESET signals to the XJSs to force them into a "sleep" mode. In order to entry this sleep mode, the system must first save any relevant system state and flush the caches. It can then safely be placed into reset mode.

When Mazda detects any reason to resume full operation, the XJSs will be taken out of reset. The reset vector will point to system code which restores the state of the system to the point at which sleep mode was entered.

An alternative is to have a "standby +5" power supply output which is just able to keep Mazda (and relevant I/O chips) alive. At "sleep" time, a DRAM image is created on hard disc which is the "wakeup" code. After writing this image to disc, Mazda would trigger the power supply to shut off all "normal" power. Whenever an (Mazda detected) event occurs which requires a wakeup, the power supply is turned on, Mazda downloads a "wakeup" DRAM startup image (different from the "reset" image) from ROM. This image initializes DRAM (quick test, parity initialization, etc.) and then reads the wakeup image from disc.

Note that the wakeup disc image can be configured to be "ready" to field any incoming data (e.g., answer the phone). Thus, the normal load time (which, hopefully is different than the Mac of today, anyway...) is bypassed for wakeup.



Apple Jaguar Manufacturing Issues

144

External Reference Specification
Special Projects

10 November 1989

Return Comments to:
Toby Farrand and John Martin
AppleLink: FARRAND1, MARTIN.J
MS: 60-E

Contents

Section 1

Introduction

Unlike the other chapters of this ERS, this chapter focuses more on *how* we plan to implement Jaguar, than *what* we plan to implement. The definition of the Jaguar machine is influenced by both top-down market and business constraints, as well as by bottom-up engineering and manufacturing constraints. In order to more fully achieve our market and business goals, Jaguar plans to push Apple's manufacturing capabilities very hard. Consequently, this chapter is devoted to briefly discussing the manufacturing technologies which Jaguar will be particularly dependent on. We do this both to justify our dependence on these technologies, and to raise the visibility of these technology developments within Apple, since Jaguar's very existence is dependent on Apple's ability to make these technologies real. There are clearly more manufacturing issues with Jaguar than just those discussed here, but this should provide an overview of the major manufacturing issues as we know them today.

As discussed previously, Jaguar will establish new price/performance points for personal computers. It will do so with very aggressive cost goals, and in a form factor that is at least as attractive as the personal computers we sell today. This means that we must put a 100MIPS multi-processor, 50MHz RISC machine in a box similar in volume to a Macintosh IIcx for a cost not much more than a IIci. We must do this without sacrificing reliability, noise or EMI constraints. There is little doubt that meeting these goals will require use of new materials and approaches to the Jaguar industrial design.

The Jaguar team's ability to produce a manufacturable design is closely related to our ability to manage the process, materials and test issues involved with the design. All these issues are discussed here.

Section 2

Process

Packaging and Interconnect

One of the key architectural features of Jaguar is its use of point to point, instead of bus oriented interconnect topologies. In the past, point to point interconnect has been avoided because it led to high-pincount devices and difficult signal routing. Another key departure for Jaguar is the fact that its data buses are all 64 bits wide, some of which are interleaved to 128 bits for higher performance. Taken together these two attributes push us toward much higher pincount packages than have characterized Apple designs of the past.

Jaguar's designs will also be clocked much faster than in previous designs and will be based on a .8μ technolog, so lead inductance will become more of an issue for both speed and EMI reasons. Power dissipation will also be an issue as most of the devices in the design are likely to dissipate over 2 watts of power.

All of these challenges point toward using TAB as our primary IC packaging technology.

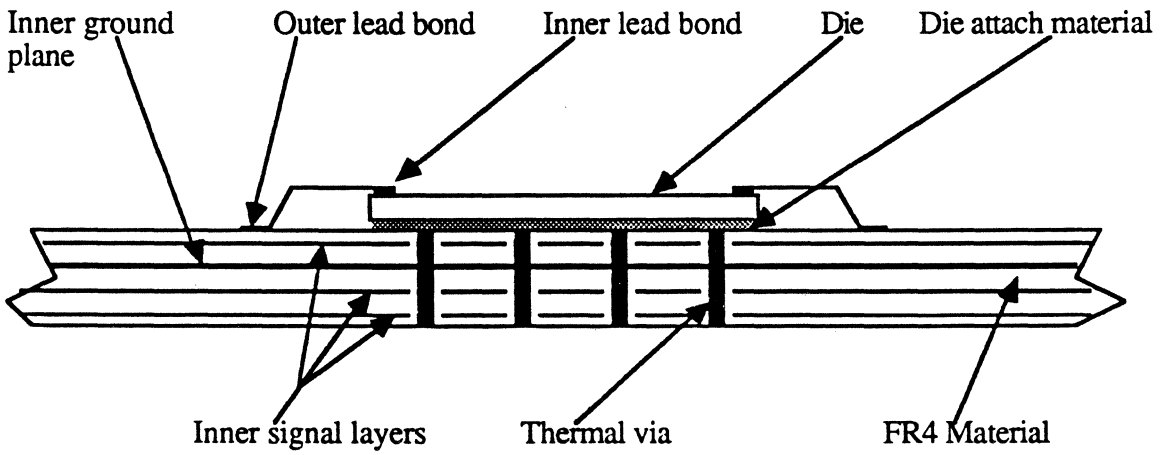


Figure 1. TAB Schematic.

Figure 1 shows a typical TAB interconnect. While the details of Jaguar's TAB process may vary, the key features of this arrangement are that the die is mounted on a thermally conductive material which glues the die to a ground pad on the PC board. This ground pad is connected to the inner ground plane of the board with thermal vias. In this way, the entire ground plane of the board is used as a heat sink for the device.

The key attributes of TAB which are attractive to Jaguar are:

- Leads have a larger cross sectional area than wire bonds and are shorter than wire bonds leading to much lower lead inductance (2nH for TAB vs. 5nH for wire bond), better signal quality and better power dissipation.
- Outer lead pitch is 15 mils (vs 25 mils for QFP packages) leading to smaller device footprint.
- Die attach process promises much better power dissipation than plastic packages through use of thermal vias into the power plane of the board. Indeed, it is doubtful that plastic packages could be used at all for many of the devices we will need on Jaguar due to plastic's poor power dissipation characteristics. Inner lead bond is 4 mils (vs. 6.5 mils for wire bonding) leading to smaller die for bond-pad limited devices.

These characteristics lead to devices with higher reliability, better EMI characteristics and lower cost than devices packaged in plastic or ceramic.

TAB is not a proven technology and carries with it some risks. The key risks in making this technology work for Jaguar are:

- Solder thickness and solderability. Until Fremont gets more experience with this process in a high volume environment, this is likely to remain a significant issue.
- Lead protection. Preliminary work in this area is promising.
- Die attach and heat dissipation characteristics. Perfecting a die attach process will be key to allowing us to avoid ceramic packages for the 88110 and other high power devices.
- Test and burn-in strategies and die qualification. TAB devices are very difficult to burn-in because of the lack of cheap reliable sockets, and the dangers of whisker grown at high temperatures on the tin leads.
- Overhead and manufacturing cost as compared to a more standard process. We need to get a better handle on exactly how much this technology will cost. We do not have reliable data yet for the capital cost, throughput, yield and labor required for a manufacturing line using this process.
- Technology timeline, especially for 10 mil OLB process. Fremont is only just beginning to get familiar with a 10 mil OLB process required for 284 pins in a 35mm tape format. This may require that we plan on using 48mm tape for our highest pincount devices.
- Cost of 48mm tape vs. 35mm. We don't have good data for the relative costs of these tape formats. If we can't make 10 mil OLB work, then we will have to depend on 48mm tape being affordable.

- Backend yields for the semiconductor vendor and overall device cost vs. QFP. The semiconductor vendors are not confident in their ability to perfect the back end of their process for TAB devices. Significant yield loss here could adversely affect the economics of TAB.

Individually, each of these risk items is manageable. Taken together, they present a great challenge. The alternative of using large numbers of ceramic pin grid arrays, or repartitioning the design to fit into Quad Flat Pack packages is unattractive for cost, space and power reasons. Consequently, we are compelled to bet on TAB.

System Configurations

Jaguar provides an opportunity for Apple to redefine the base capabilities to be expected in a personal computer. This presents some unique benefits and challenges from a system assembly and packaging point of view.

Every member of the Jaguar family will have a hard disk built-in. This means that system software can be configured onto each disk, and the in-box materials need not include floppies. It isn't clear what this implies from a manufacturing point of view. The Macintosh Portable currently ships with system software configured on its hard disk, so it doesn't appear that this should present any unique problems.

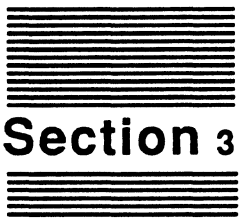
Jaguar will have a way of providing a unique serial number for each machine built. This number will be stored either on the internal hard disk or in the same EEPROM used to store the system boot code. This will allow easier tracking of machines in the field and should provide some opportunities for service. This will require some process during manufacturing to create and load this serial number into the machine.

Jaguar's networking and telecommunications strategy requires modules external to the machine to customize Jaguar to the customer's Ethernet environment and to the phone requirements of whatever country the customer is in.

As described in the I/O section of the ERS, Jaguar will support Ethernet via the Apple Attachment Unit Interface (AAUI) which will connect to "thin net", "thick net" and twisted pair ethernet by way of a Media Access Unit (MAU) external to the box. We may choose to bundle the appropriate MAU with the machine to afford easier "plug and play" for our Jaguar customers.

We will also be borrowing from the efforts on the Data Access Arrangement (DAA) developed for the Macintosh Portable. This will allow the main Jaguar unit to have a single telecom interface that is then customized via the DAA appropriate for the target country. Here too it is likely that we will want to bundle the DAA with the Jaguar.

At minimum, this strategy raises significant localization issues.



Materials

Component Strategy

Jaguar's ambitious cost and performance goals can only be met with state-of-the-art ASIC and board interconnect technology.

All Jaguar ASICs and the XJS processor will use a .8µ CMOS technology. As described above, several ASICS will be packaged in high-pincount TAB packages. This aggressive use of technology raises several materials issues ranging from qualification of the .8µ process, to qualification of TAB, and management of unique test and burn-in issues TAB raises. Few problems are anticipated in qualifying the .8µ process we are targeting as it is based on a 1Mb DRAM process which is currently shipping in volume. There are a host of issues with TAB which are being addressed by the TABBing project which is focusing on the basic issues involved with manufacturing, testing, burning in, and bonding of the TAB device. There we are also beginning a project to evaluate TAB's power dissipation characteristics with a die attach process.

Another area where Apple will be pushing technology very hard is in the PC board area. Current state-of-the-art for Apple is a 6 mil line, 6.5 mil trace technology. It is likely that Jaguar will be forced to push this technology as well. Programs to develop this area are not so far along as the TAB program, but are very important. Soon we will begin to work these issues as well and put in place programs to push to 5/5 and 3/5 technologies.

Vendor Strategy

In addition to the technical issues involved with TAB are issues around supply base management and procurement for these unique technologies. Very few vendors have the technology and capital required to provide for our TAB needs. Furthermore, Apple is limited in its ability to work with vendors to bring these technologies to production. Consequently, Jaguar is working with a single vendor to perfect TAB and to provide for all of our ASIC needs.

This single vendor strategy has the advantages of letting us focus our resources on perfecting a single process, and it allows us to simplify our ASIC design environment because one set of libraries and tools are used for the entire machine. The disadvantages are that we are totally dependent on one manufacturer to support us. This is further complicated by the uncertain nature of Jaguar's procurement needs. It is very difficult to predict how rapidly Jaguar will gain acceptance in the market because of the tradeoff our customers will have to make between getting a quantum leap in price performance and new features, and moving to a new and unfamiliar architecture. This

uncertainty translates into significant risk in terms of capacity planning for a single vendor to manage.

To help manage these risks, the Jaguar project will be working much closer with our ASIC vendor than Apple has typically worked in the past. The vendor will have one or two of their employees on site at Apple, and we are cultivating a unique relationship in which we have very detailed information on their cost structure such that we can anticipate their business needs and take advantage of synergies between the Jaguar project and the vendor's capabilities. To further protect Jaguar from the risks of working with a single vendor, we are negotiating for the vendor to support a second source for their ASIC process within a year of Jaguar's ship.

Jaguar is negotiating a contract with our ASIC vendor to clearly define our relationship, and make clear what is required from the vendor in order to satisfy Jaguar's requirements.

While Jaguar is well along in working the vendor issues related to ASICs, much more work is needed to manage the issues related to the PC board interconnect technology we will require. This is likely to require a new and more intimate relationship with a small number of PC board suppliers to meet our quality, quantity, cost and turnaround goals. Work in this area has not progressed much so far, but it will gain increasing importance as the project progresses.

Product Design Issues

Product design has some ambitious plans for innovative ways to cool Jaguar and make the machine as elegant and unobtrusive as possible. This will doubtless involve new materials and processes for producing the Jaguar enclosure. No decisions have been made to date as to what technologies are required, but it should be noted that significant issues will likely come up in this area.

Section 4

Design for Testability

Introduction

At the board level, it is clear that the density of interconnect and TAB usage demanded by Jaguar will make a traditional bed of nails tester impractical. A limited number of test points at best will be available on digital real estate (analog test points are required and will be included). Consequently, Jaguar will rely heavily on the standard test access port (TAP) architecture developed by the Joint Test Action Group (JTAG) and further modified by IEEE P1149.1 test standards committee to implement its Design for Testability techniques, most notably boundary scan. Boundary Scan is merely a continuity test; it tests for pin faults caused by such failures as solder opens and lead breakage, thus *all ASICs must be fault graded for 98+% fault coverage.* Although internal tests can be supported by P1149.1, they are expensive due to the limitation of serial access. Thus the commitment for reliable ASIC's are a must if we limit board test access. The goal is to allow only 30-45 seconds for board test so that test is not a production bottleneck. To further that goal and allow for internal test for rework and initial debug, an optimal set of scan chains must be derived rather than a simplistic unary scan chain. Interestingly enough, if operating system hooks are included, a limited self testing system is a possibility.

Objective for In-Circuit Testability:

Standardization. The architecture should codify many external and internal "Design For Test" (yet another acronym, DFT) techniques into a standard vector instruction format. ASIC designers should not need to create their own flavor of boundary scan architecture.

Shortened Test Development time. The architecture's consistent board level interface should allow a post-processor that recognizes the scan chain topology to easily create serial test programs.

Increased Board Density. The risk of increasing system board size is driving Jaguar designers to be less silicon frugal in their ASIC designs. Because the designs are core limited, test circuitry will take up less percentage of silicon area. Thus the board area expensive test points can be discarded for density afforded by TAB technology.

The IEEE P1149.1 TAP architecture provides the necessary basis to achieve the above objectives in each ASIC. XJS is specified for JTAG compatibility; since P1149.1 is a proposal standard that should be ratified in the upcoming weeks, we must confirm that Motorola intends to update its specifications.

TAP Architecture for Internal and External Test

The TAP architecture allows the following minimum implementation: four pin test access port, a serially fed instruction register, a serial shift register which traverses the chip's boundary nodes, and a single bit bypass register which allows the chip's serial scan path to be selectively avoided during board level testing. Optionally, there is a provision for including a device identification register and internal scan loops for such internal tests as Level Sensitive Scan Design (LSSD) or Built-In Self Test (BIST). The TAP controls the operation of these instruction and data registers via a 16 state finite state machine known as the TAP controller.

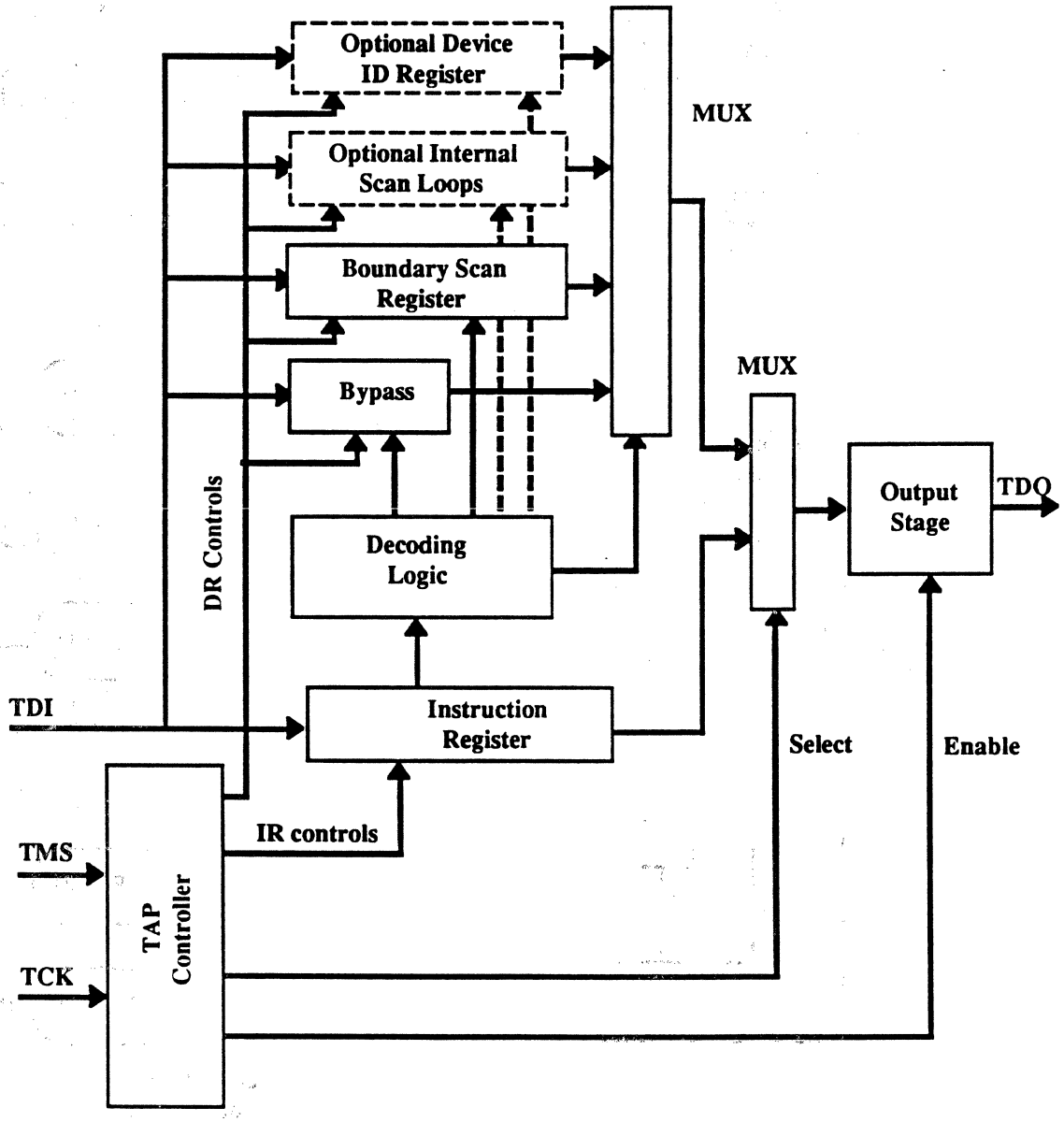


Figure 1 TAP Architecture

Figure 1 shows a block diagram of the TAP Architecture. The TAP controller is fed by two package pins known as TCK (Test Clock) and TMS (Test Mode Select) which drives its state machine. A diagram showing the state transitions of the TAP controller can be found in Figure 2. These transitions follow the logic value asserted on TMS at the rising edge of TCK. Notice that the state transition table has been designed in such a way as to guarantee resetting the machine (a return to Test Logic Reset) after five assertions of a logic 1 at TMS.

The state machine essentially performs three functions: instruction register scan, data register scan, and test execution. During instruction register scan, data which appears at the TDI (Test Data In) pin during the rising edge of TCK is shifted into the serial instruction register. The bits in this instruction register are then decoded to select the test instruction to be performed, as well as deciding which data register (bypass, boundary scan, or user defined) will be accessed by a subsequent data register scan. Data register scan operates in much the same way, shifting TDI data into the currently selected data register at the rising edge of TCK. Both instruction and data register scans will shift data out on the TDO (Test Data Out) pin. TDO will be valid after the rising edge of TCK, and is tri-stated except when data is actually being shifted out. During most scan operations these TDO values will be taken from the currently selected test register. The third function of the TAP controller is to execute tests. Once the instruction for a test has been shifted into the instruction register and the appropriate data register has been selected, looping around the "Run Test / Idle" state causes the test to occur. Test results would then be shifted out by doing a data register scan.

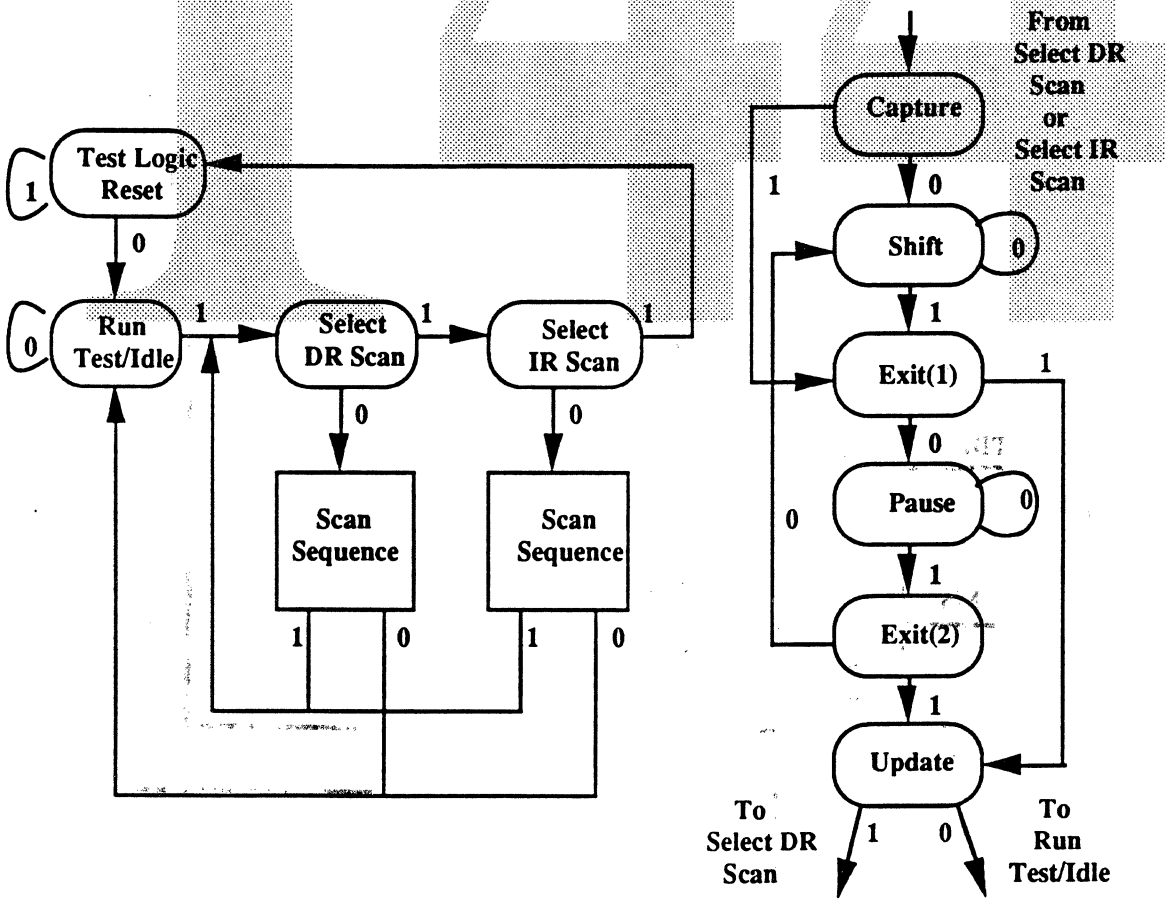
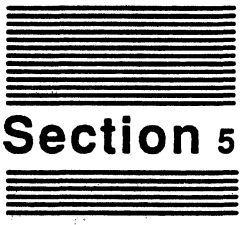


Figure 2. TAP Controller State Diagram

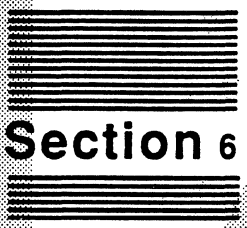
Work Remaining

Obviously we must work in concert with the IC Vendor to assure scan cells and TAP architecture is optimized for our purposes (i.e. fulfill timing expectations and board test requirements). The TAP architecture must be rigorously adopted by all ASICs as well as XJS. A well defined board topology is needed (with associated data paths if an internal test is performed) to derive an optimum set of scan chains in order to complete board test in a sane time frame. Then, an exhaustive set of continuity tests can be derived. A test vector post-processor must be created or found to serialize and pad test streams. ASIC's must be fault graded for 98+% fault coverage.



Section 5

ASIC Test Plan



Section 6

Board Level Test Plan