

Hoover Bus: An Input Bus Optimized for Multiple Real-Time Data Streams

by

Jeffrey J. Wong

B.S., Electrical Engineering
Massachusetts Institute of Technology (1996)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

September 9, 1996

© 1996 MIT. All rights reserved.

Author _____

/ Department _____

Computer Science
September 9, 1996

Certified by _____

V. Michael Bove Jr.
Associate Professor of Media Technology
MIT Media Laboratory

Accepted by _____

F.R. Morganthaler
Department Committee on Graduate Thesis

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

OCT 29 1997

ENG

LIBRARIES

Hoover Bus: An Input Bus Optimized for Multiple Real-Time Data Streams

by

Jeffrey J. Wong

B.S., Electrical Engineering
Massachusetts Institute of Technology (1996)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

September 9, 1996

ABSTRACT

The growth in demand for high bandwidth, real-time multimedia applications has resulted in the development of several bus standards in the computer industry. Differing design objectives, operating environments, and cost constraints have yielded many designs with varying engineering tradeoffs and optimizations.

This document covers the design and implementation of the Hoover Bus, an inexpensive, dedicated input bus optimized for the transmission of multiple real-time virtual channels of media across a single physical channel. By carefully limiting the scope of the expected applications of this bus protocol, optimizations can be made in the datapaths, arbitration mechanisms, and flow control mechanisms which increase usable throughput in heavy bus traffic conditions. A test configuration of the Hoover Bus will be constructed, and a detailed specification delineating the design tradeoffs and their motivations will be constructed to aid the further development of this standard.

Thesis Supervisor: V. Michael Bove Jr.

Title: Associate Professor of Media Technology

This work was supported by the Television of Tomorrow consortium.

This thesis is dedicated to Moy-Shee Chin (1904-1996)

ACKNOWLEDGMENTS

This thesis would not have been possible without the help and support of many people. First and foremost, I would like to thank my parents and family, who have encouraged, guided, and supported me through my life. Without them and their love, I would have never made it this far.

Next, I would like to thank my thesis advisor, Professor V. Michael Bove Jr., and John Watlington – a true EE/CS maven and friend. Since my freshman year, they have enriched my MIT education far beyond the normal EECS coursework by giving me projects and opportunities which few MIT students see. They taught me concepts and tricks which proved far more useful than some of the complex “book smarts” taught in classrooms, and kept me interested and aware of the latest and greatest in the EE/CS field.

Matt Antone, my academic symbiote and good friend, deserves more credit than I can articulate on this page. Over the past five years, we have taken the same classes, taught each other, and enriched each others’ lives through many discourses on countless subjects, both academic and non-academic. It has been a blessing to have his friendship for the past five years.

A salute and smile also are extended to Allen Sun, Ross Yu and the other members of the Cheops crew. Allen saved me countless hours by constructing the slave units from my schematics, and provided valuable feedback during the design stages. Ross and I share the special distinction of being veterans of the Cheops corner of the Garden - a mystical place where mere mortals try to influence the path of electrons and information through design, exasperation, and prayer.

I would also like to thank the people who provided the logistical support to get me through the last, most intense portions of this project. Matt and Yuli deserve much thanks for giving me a temporary place to live; I also would like to thank Lucy, Scott, and the other residents of Conner 2 for giving me another “home base” on campus.

There are so many more people whom I would like to acknowledge: Sue, Sonia, Nancy, Jung, Dave, Di, the Garden, the S/E program at MHS...the list would go for pages. All I can say is, you know who you are; each of you is very special to me, and I thank you for everything.

Table of Contents

Chapter 1: Introduction	11
1.1: Overview	13
Chapter 2: Previous Works	16
2.1: Ultra SCSI	16
2.2: IEEE P1394	19
2.3: Scalable Coherent Interface	22
Chapter 3: Bus Architecture	24
3.1: Data Transfer Protocol	24
3.1.1: Electrical Signalling Layer	24
3.1.1.1: Design Objectives and Design Process	24
3.1.1.2: Helpful Techniques	25
3.1.1.3: Differential Signalling	25
3.1.1.4: Lower Signal Amplitudes and Limited Slew-Rate Signals	26
3.1.1.5: Termination of Signals	27
3.1.1.6: Synchronous Data Transfer	28
3.1.1.7: Backplane Transceiver Logic	29
3.1.2: Bus Signal Definition	30
3.1.3: Data Packet Definition	31
3.1.3.1: Packet Size Design	32
3.1.4: Arbitration Scheme	34
3.1.4.1: Slave Arbitration FSM - Primary (“Greedy”) Model	35
3.1.4.2: Slave Arbitration FSM - “Generous” Model	39
3.1.4.3: Other Variations of the Arbiter FSM	40
3.1.4.4: Examples of Hoover Bus Timing	41
3.1.4.5: Simple Hoover Bus Timing	41
3.1.4.6: Greedy Slave Arbitration Timing	42
3.1.4.7: Generous Slave Arbitration Timing	43
3.2: Command Protocol	44
3.2.1: Electrical Layer	44
3.2.2: Protocol Design Considerations	46
3.2.3: Messaging Protocol	48
3.2.4: Existing Serial Protocol Opcodes	49
3.2.5: Structure of Master to Slave Serial Messages	50
3.2.5.1: Encoding of Numerical Arguments	51
3.2.6: Structure of Slave to Master Reply Message	51
3.2.7: Flow Control	52
3.3: Slave Design Overview	52
3.3.1: Data Producer	53
3.3.2: FIFO Buffer	53
3.3.3: Consumer	54
3.3.4: Bus Interface Unit	54
3.3.5: MCU and Control Logic	57

3.3.6: Producer - FIFO Interface and Overflow Mechanisms.	57
3.3.6.1: Overflow Handling with Single Stream Producer Outputs	58
3.3.6.2: Interfacing the Producer and FIFO - Multiple Symmetric Outputs	58
3.3.6.3: Interfacing the Producer and FIFO - Multiple Asymmetric Outputs	59
3.4: Bus Master Overview	61
3.4.1: Bus Interface	61
3.4.2: Data and Header Registers	61
3.4.3: FIFO Block	62
3.4.4: Arbiter	62
3.4.4.1: Error Checking State Machine	65
3.4.5: Clock Circuitry	65
3.4.6: Address Generator	65
Chapter 4: Implementation	67
4.1: Slave Implementation	68
4.1.1: Slave Behavioral Description	69
4.1.2: MCU and Serial Interface	69
4.1.2.1: Slave Core Controller Program	70
4.1.3: Producer/FIFO Buffer Status Indicators	73
4.1.4: Consumer and FIFO Data Circuitry	74
4.1.4.1: SRAM Pattern Buffer	74
4.1.4.2: SRAM Address Generator.	76
4.1.4.3: Producer and MCU SRAM Control Circuitry.	77
4.1.5: Arbiter/Upstream Clock Convertor/Downstream Clock Convertor	78
4.1.6: Bus Interface Unit.	79
4.2: Bus Master Implementation	80
4.2.1: Memory Store Unit	80
4.2.2: Address Generator	80
4.2.3: Bus Interface Unit	82
4.2.4: Bus Master Arbiter and Clock Circuitry	82
4.2.5: FIFO Module	82
4.2.6: Serial Interface	83
4.3: Software Interfaces	83
4.3.1: Serial Port Interface Software	83
4.3.2: Uploading Slave Core Controller Code	85
Chapter 5: Conclusions	87
5.1: Current State of the Test Bus Configuration	87
5.2: Evaluation of Current Design and Recommendations for Future Work	88
Appendix A: Bus Transfer Timing Calculations	91
A.1: Average Handshaking Overhead	91
A.1.1: Average Overhead for Typical Bus Configurations	92
A.1.2: Worst Case Average Arbitration Time	92
A.2: Maximum Packet Interarrival Rate	93
Appendix B: Termination Techniques	95

Appendix C: The Cheops Imaging System.	99
Appendix D: 6811 Utility Source Code	101
D.1: Source Code for convertor.c	101
D.2: Source Code for MCU EEPROM Programmer Routine	103
Bibliography	108

List of Figures

Figure 1-1: Comparisons of Serial and Parallel Topology Buses.	14
Figure 2-1: Arbitration for (Ultra) SCSI.	17
Figure 2-2: Bit-strobe encoding	20
Figure 2-3: Normal FireWire Arbitration.	20
Figure 2-4: Isochronous Arbitration	22
Figure 3-1: Environment for Crosstalk Measurement	27
Figure 3-2: Structure of a Data Packet	31
Figure 3-3: Primary State Machine for Arbiter.	36
Figure 3-4: Generous Arbiter FSM.	40
Figure 3-5: Overview of Simple Hoover Bus Timing.	41
Figure 3-6: Greedy Slave Arbitration Timing Diagram	42
Figure 3-7: Generous Arbitration Timing.	43
Figure 3-8: RS-485 Four Wire Multidrop Network	45
Figure 3-9: Structure of Master to Slave Message	50
Figure 3-10 Structure of Slave to Master Reply Message	52
Figure 3-11: Block Diagram of Slave Unit.	53
Figure 3-12: Expanded Block Diagram of Slave Unit	55
Figure 3-13: Slave Node Arbiter Signals	56
Figure 3-14: Overflow Handling via Auxiliary Pre-FIFO	58
Figure 3-15: Packet Flushing Strategies for Producers with Symmetric Outputs.	59
Figure 3-16: Bus Master Block Diagram	61
Figure 3-17: Bus Master Arbiter.	63
Figure 3-18: Bus Master Arbiter State Diagrams	63
Figure 3-19: Hoover Bus Master Arbiter Timing.	65
Figure 4-1: Test Slave Block Diagram	68
Figure 4-2: Outline of Core Controller Program	71
Figure 4-3: Register Map of Test Slave Unit	72
Figure 4-4: Hoover Test Slave SRAM Buffer Diagram.	76
Figure 4-5: Slave Termination Circuitry.	79
Figure 4-6: Address Generator and Support Circuitry	81
Figure B-1: Different Types of Termination.	95
Figure C-1: Block Diagram of P2 Processor Card	99

Introduction

Chapter 1

One of the fundamental design issues faced by digital multimedia system architects is how to transfer large amounts of time-critical data between components in the system. Sources such as digital video cameras can require bandwidths upwards of 26 MB/s, and currently, it is not unthinkable for a video processing computer system to require several such input sources for certain applications.

Prior to the explosive growth of multimedia applications in personal computers, peripheral buses with sufficient bandwidth for digital multimedia were non-existent, except for a few specialized, expensive buses such as HIPPI which were designed for high performance computing.

However, as multimedia, video conferencing, and digital video became buzzwords in the computer industry, system designers required buses that could handle such bandwidth-intensive applications. In response, bus standards such as SCSI (Small Computer

System Interface) were re-engineered to faster incarnations such as Ultra-SCSI. In addition, a new generation of higher bandwidth bus standards were developed, including Fibre Channel, Scalable Coherent Interface (SCI), and FireWire (IEEE 1394).

At the MIT Media Lab, the Television of Tomorrow group also faced this need. Applications for the Cheops Real-Time Image Processing System, a modular dataflow architecture designed by Professor V. Michael Bove Jr. and John Watlington for experiments in structured video and spatial imaging [1], required the real-time transmission of several virtual channels of media. However, it was also determined that none of the existing general-purpose peripheral buses were well-suited for these applications except for a few more expensive and less readily available standards, such as FiberChannel and SCI.

As a result, the TVOT group proposed the design and implementation of a simpler, specialized bus which was optimized to handle multiple virtual channels of real-time traffic. By limiting the expected bus traffic to media streams, much of the overhead necessary for the flexibility of general purpose buses could be removed. In addition, mechanisms such as arbitration which typically favored the more common random bus traffic case, to the detriment of real-time data transmission, could be redesigned to meet the demands of transporting media streams.

A set of design parameters were developed for this proposed stream-oriented bus, including:

- high throughput capacity - 80 MB/s.
- capable of handling up to eight external, physically separated slave devices per bus
- capable of handling multiple virtual channels of data from each slave unit
- graceful degradation under heavy load
- low cost
- non-centralized bus arbiter to avoid wiring overhead
- easy to implement via conventional construction techniques - no ASICs or extensive

surface mount work needed for initial slave prototypes.

1.1 Overview

The Hoover Bus is the proposed design which met all of above requirements. It consists of one or more data-producing slave units linked to a single bus master in a broken token-ring topology with the bus master at one end of the chain, interfacing the bus to a large memory store. Slave units can be physically separated units, connected together with standard SCSI cables containing 16 data lines, a data clock, two arbitration lines, and four serial communication lines.

The bus can support up to 255 independent virtual channels of data from the eight slave units. Data transfers are unidirectional, running *downstream* from the *active slave* (the slave which is actively transferring data) to the bus master in 514 byte packets consisting of a two byte header and 512 data payload bytes.

The bus master interfaces the Hoover Bus datapaths to a large memory store via a FIFO bank, arbitrating internally for access to the memory with other bus controllers. The bus master extracts stream identification information from each packet header and calculates the appropriate address for the memory store to use while storing the data packets.

Master-slave communication is achieved through the serial bus, which uses the RS-485 signalling protocol. The master initiates all commands; slaves can only send reply messages and error messages when prompted by the master. Microcontroller units on each slave unit independently determine if the message applies to them and take the appropriate followup action.

Unlike parallel topology buses such as SCSI-2, data transfers on the Hoover bus are made through point-to-point links between slave nodes. Each slave actively retimes

and retransmits incoming data to allow a longer bus length and higher data transfer rates (see Figure 1-1). The active slave generates data and a 40 MHz dual-edged data clock signal and sends them through the connecting cables using Backplane Transceiver Logic (BTL). The active slave's downstream neighbor receives this signal, translates it to TTL levels and uses the clock edges to latch an internal resynchronization register which then outputs the data and regenerated clock on the slave's downstream port to the next slave unit in the bus.

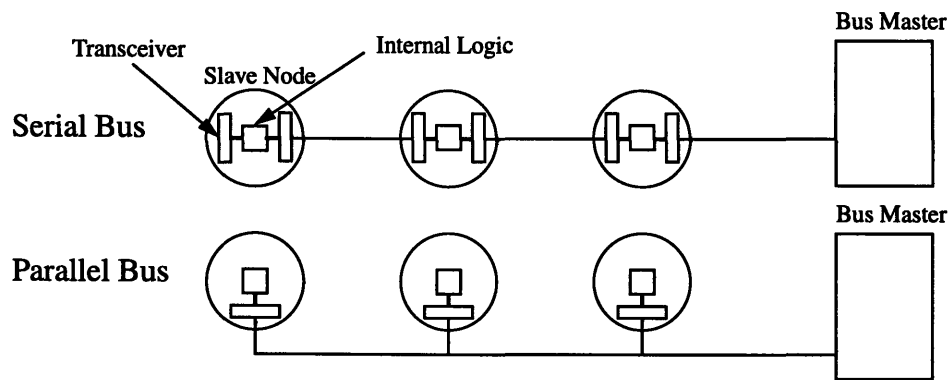


Figure 1-1: Comparisons of Serial and Parallel Topology Buses

Slave units arbitrate for bus access via two arbitration lines, ACK and REQ, using a standard four phase handshaking protocol. To support graceful stream degradation, the Hoover Bus uses a fixed, biased arbitration scheme which gives higher priority to slave nodes closer to the bus master. With this strategy, if demand for bandwidth exceeds bus capacity, the downstream slaves whose bus demand is supportable by the available bandwidth will be able to transfer their data streams with minimal interruption, while all slaves further upstream will have their streams decimated.

Overflow errors in a stream are guaranteed to be the loss of an integral number of packets, and are recoverable through synchronization information contained in the data

packet headers. Bus designers have the option of implementing flow control mechanisms in software which enable the bus master to automatically reduce the throughput of the lower priority slave media sources if overflow errors occur.

This thesis will step through the development and implementation of the Hoover Bus. Chapter two will examine the contemporaries of the Hoover bus and discuss the relative strengths and weaknesses of their designs relative to our desired applications. Chapter three will step through the theoretical design behind and specifications of the various subsystems in the Hoover Bus. Chapter four will describe the first implementation of a Hoover Bus - two test slaves which can emulate the functionality of a variety of slave devices and a bus master on the Cheops system which interfaces to a 4 GB DRAM store. Chapter five will analyze the performance of the Hoover Bus test system and examine directions for future research.

Previous Works

Chapter 2

Three interface standards - Ultra SCSI, FireWire (IEEE 1394), and Scalable Coherent Interface - with design goals similar to the Hoover Bus were studied in order to better understand the “state of the art” and the current approaches used to increase bus performance.

2.1 Ultra SCSI

Ultra SCSI (formally known as Fast-20 SCSI) is a part of the SCSI-3 standard drafted by the American National Standard XT310 Technical Committee. It defines electrical and mechanical improvements which double the maximum throughput of SCSI-2 buses (see Table 1 below) while maintaining backwards compatibility with existing equip-

ment [2] [3].

Table 1: Maximum Throughput of Various Versions of SCSI

Version	Number of Signal Wires	Maximum Throughput
SCSI (SCSI-1)	8	5 MB/s
SCSI-2 (Fast SCSI)	8	10 MB/s
Wide SCSI-2 (Fast Wide SCSI)	16	20 MB/s
Fast-20 SCSI (Ultra SCSI)	8	20 MB/s
Wide Fast-20 SCSI (Wide Ultra SCSI)	16	40 MB/s

The Ultra SCSI bus is a parallel topology bus which, in its single-ended signal design, allows up to eight devices to be connected in a linear chain with 50 conductor or 68 conductor cable [2]. Devices can be a variety of types, including disk drives, CD-ROMs, processors, and sequential read devices. Both asynchronous transfers and faster synchronous transfers are supported. Distributed arbitration is accomplished through bus-contention logic to simplify wiring (see Figure 2-1), and a rich command set allows the bus master to initiate a whole range of actions on a slave or obtain configuration and status information from a slave device [5].

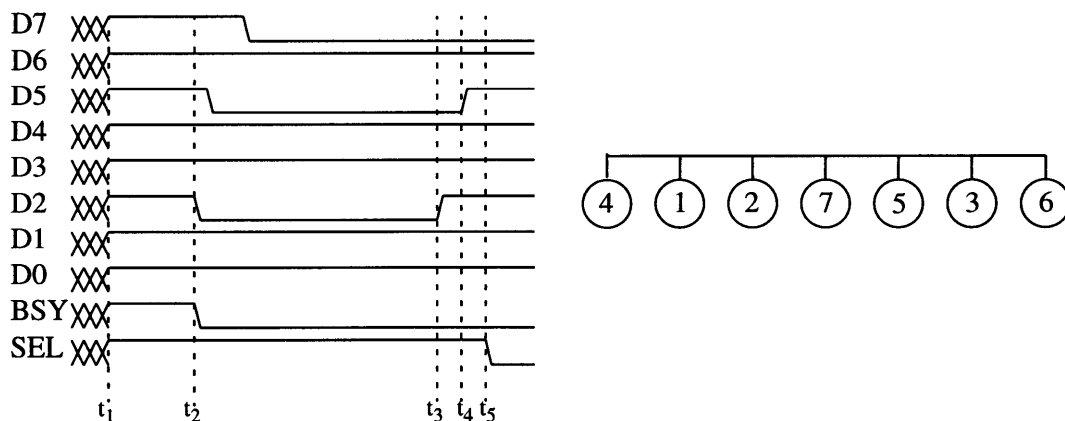


Figure 2-1: Arbitration for (Ultra) SCSI.

The diagram on the left shows the state of the data and control signal lines for SCSI device

#2 from the corresponding SCSI bus on the right. During the bus free phase from time t_1 to time t_2 , node 2 waits to confirm a lack of activity on the bus. At time t_2 , the device asserts BSY and D2 to arbitrate for the bus. Due to signal propagation delay, devices 5 and 6 fail to see device 2's request for the bus, and they request bus access as well. At time t_3 , device 2 has waited for an arbitration delay, it checks the data lines and realizes that another device with a higher ID has requested the bus, so it deasserts its line. At time t_4 , device 5 realizes that it, too, has lost the arbitration, and deasserts its line. At t_5 , device 6 has waited for an arbitration delay, and upon checking the bus, realizes that it has won arbitration. It asserts SEL, and proceeds to the selection phase

However, several characteristics prevent Ultra SCSI from being an optimum solution for a dedicated real-time input bus. First, because compatibility with existing SCSI devices was a prime requirement, the parallel topology of SCSI-1 and SCSI-2 had to remain unchanged, resulting in lower signal speed and integrity when compared with new generation serial topology buses.

Second, since it is derived from SCSI-1 and SCSI2, Ultra SCSI retains the protocol overhead necessary for a general-purpose bus. For example, a device to device transfer consists of several distinct steps: bus free phase, arbitration phase, selection phase, reselection phase, and information transfer phase (which is one or more of the following: command phase, data phase, status phase, and message phase) [5]. Some of these phases, such as the selection phase which designates the target of a transfer [5], are unnecessary in the data transfers for a dedicated input bus, which only has one target for data transfers.

Third, in order to maintain compatibility with slower, older devices, the timing of these phases is kept longer than what is implementable with current technology; arbitration periods, for example, must last 2.4 μ s even though today's output drivers could probably operate correctly with significantly shorter periods.

Fourth, a significant portion (eight wires) of the available wiring resources in the Ultra SCSI bus are used for control signals such as REQ, BSY, SEL, and I/O. Although

bus state can be unambiguously determined from these lines, this approach also wastes bandwidth - a real limitation since the maximum throughput of wide Ultra SCSI without interruptions for commands and messages is already less than the desired bandwidth for our input bus.

2.2 IEEE P1394

IEEE P1394, commonly known as FireWire™, is a high speed bus whose primary design goals included low cost, high bandwidth, ease of use, and design of both backplane and external cable variants [7]. The cable variant, which is more applicable for the requirements discussed in Chapter 1, supports data transfer rates of 98.304 Mbit/s, 196.608 Mbit/s (2x), and 393.216 Mbit/s (4x) [6]. The bus is a serial topology bus which supports up to 63 slaves in a non-cyclic point-to-point network constrained to the conditions that (1) any two nodes are connected by up to 16 cable hops and (2) the maximum length for any cable hop is less than 4.5 meters [7]. To reduce connector and cabling costs, the cables are also serial in nature; only six conductors are used: two differential signal pairs, and one power pair [6].

In order to support such a rapid signalling speed with only two signals, FireWire uses Data-Strobe bit level encoding [6]. The data and strobe signals are encoded with Non Return to Zero (NRZ) levels, and the strobe signal only transitions when the data line is kept constant between two consecutive data periods (see Figure 2-2). As a result, the two signals together will always have one transition each clock period. This ensures that timing information can be obtained by the receiver each clock cycle, and that the minimum number of transitions possible are being used - increasing skew tolerance and data trans-

mission rates [6] [7].

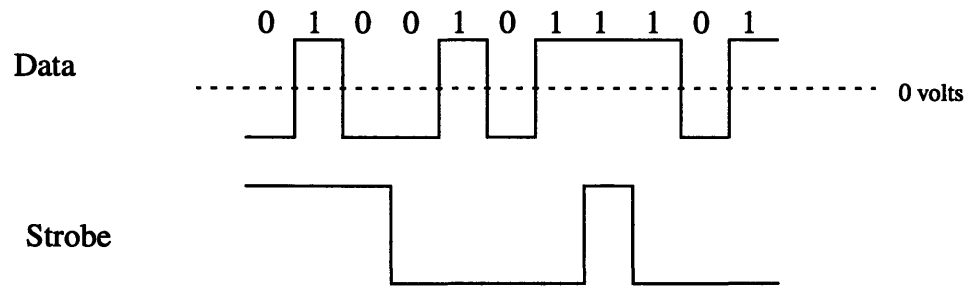


Figure 2-2 - Bit-strobe encoding

FireWire was designed to simultaneously handle both asynchronous and isochronous bus transfers [7]. In the earliest stages of design, the bus architects realized that a normal arbitration scheme with no fairness mechanisms would allow the closest node to the root to dominate the bus (see Figure 2-3). As a result, they introduced *fair* and *isochronous* arbitration mechanisms¹ [7].

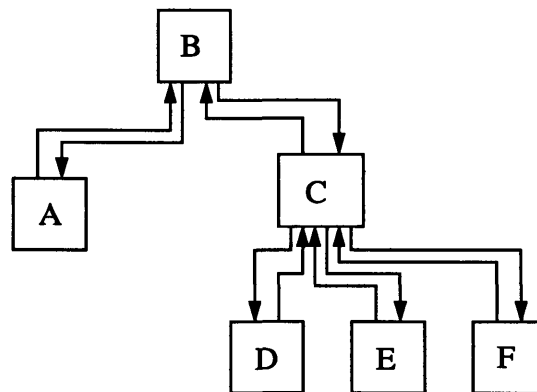


Figure 2-3: Normal FireWire Arbitration.

(Condensed from *New Technology in the IEEE P1394 Serial Bus - Making it Fast, Cheap and Easy*) If nodes A and E want access to the bus, they will wait until the bus has been idle for a certain period of time (the subaction gap) and then place requests upstream towards node B (which has been designated as the root of this tree during bus initialization). Node C will see E's request and send a blocking message to its other children, D and F while forwarding E's request to B.

1. There is also an urgent arbitration mechanism which is used for the backplane version of FireWire.

Meanwhile, Node B will see A's request first and send a grant to node A. Node B will simultaneously send a blocking message to C, which will withdraw its forwarded request and send blocking messages to nodes D, E, and F, causing E to end its request.

A will then see B's grant and then send a blocking message prior to sending out the actual data. Once the data has been sent successfully, A will go idle for a brief interval, and B will send an ack packet back to A while deasserting the blocking signal to the other subnodes. After the entire bus has been idle for another subaction gap, all slaves with data will begin arbitration for the bus again.

In fair arbitration, the bus uses fairness intervals to regulate arbitration. Each node has an internal enable register which is set true by a period of bus inactivity, the *arbitration reset gap*, which is significantly longer than the subaction gap. As each node arbitrates for, and receives access to the bus, it clears its enable register, and cannot arbitrate for the bus for the remainder of the fairness interval. When each device has successfully arbitrated for the bus once, the bus remains inactive for the arbitration reset gap, causing all the nodes' enable registers will be reset, and the next fairness interval begins [7].

In isochronous arbitration (see Figure 2-4), one predesignated node, the "cycle master", attempts to send a timing message every 125 us. If this message is blocked by a presently ongoing transaction, the master retries until the bus is clear for a subaction gap, and includes the delay value in the timing message. All isochronous nodes then arbitrate for the bus, separating their bus slots by smaller isochronous gaps which asynchronous nodes fail to see. When all isochronous nodes finish, the bus goes idle for a subaction gap, which allows the asynchronous nodes to begin arbitration for the bus. 125 us later, the

cycle master attempts to send a timing message again, repeating the cycle [7].

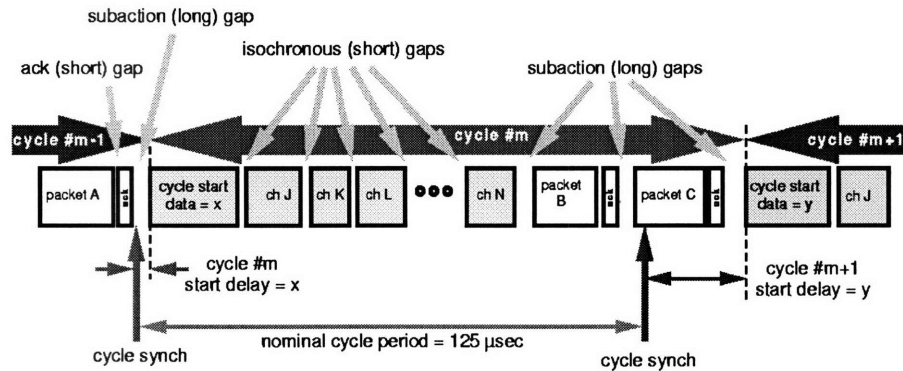


Figure 2-4: Isochronous Arbitration

(from *New Technology in the IEEE P1394 Serial Bus - Making it Fast, Cheap and Easy*)

Although FireWire is a low cost, high speed serial bus with well-designed arbitration schemes, two issues prevented us from selecting this bus for our applications. First, the maximum throughput variant of FireWire (4x) still had insufficient bandwidth to meet our specifications. Second, a lack of P1394 hardware at the time would have made initial implementations of Hoover bus devices extremely difficult. Designing control logic in user programmable devices such as EPLDs or PALs which could operate at such high speeds would require extremely expensive high-density, high-speed devices and very careful design. In addition, any prototyping with wirewrap would be simply impossible at those speeds.

2.3 Scalable Coherent Interface

Scalable Coherent Interface is a very high speed interconnect protocol that uses point-to-point links to achieve throughputs of 8 gigabits/second in local configurations. Designed by several Futurebus+ architects to be an interface suitable for connecting memory systems of 1000 processor supercomputers, SCI was designed to be fast, flexible interface which blurs the distinctions between I/O buses and high speed memory buses [8].

In SCI, data is sent between nodes with either 18 bit unidirectional wires (16 data bits, one 500 MHz NRZ clock, and one flag for packet synchronization) using ECL signaling for a throughput of 1000 MB/s or with single conductor coax cable or optical fiber run at a signalling speed of 1250 MHz for a throughput of 125 MB/s [8].

SCI supports many topologies, from meshes to rings and crossbars. 64K nodes can be in used in one logical network, and the span of the network can reach multiple kilometers with the optical fiber physical layer [8].

In order to prompt servicing of time-critical data, four priority levels are used to differentiate bus traffic. The highest priority is granted 90% of the usable bandwidth, and the remaining bandwidth is dedicated to the remaining, lower priority requests. Fairness and latency reduction techniques such as priority inheritance and queue-entry reservation mechanisms ensure that the network stays fast and fair for high priority traffic [8].

However, two of the SCI architects admit that this base protocol has “a weakness when applied to real-time applications, for which prioritized transactions require priority-dependent minimal bandwidth and maximum latency guarantees. [9]” As a result, multiple conflicting proposals on how to modify SCI have arisen, including SCI/RT-1995 and the P1596.6 SCI Specification for RealTime Applications (D0.131). Gustavson admits that SCI/RT is “an evolving work [10]” and, as a result, it was determined that although SCI/RT could be an optimum solution for our applications in the future, it is not a presently a suitable solution for our needs. Nevertheless, with its emphasis on high throughput, low latency and flexible networks, the SCI protocol remains a good source of information for bus development and innovation.

Bus Architecture

Chapter 3

The bus architecture is broken into four discrete sections: data transfer protocol, command protocol, slave design, and bus master design.

3.1 Data Transfer Protocol

The data transfer protocol covers the arbitration for and transfer of packets between the slave devices and bus master. It is broken down into four sections: electrical signalling layer, signal definition, packet definition, and arbitration protocol.

3.1.1 Electrical Signalling Layer

3.1.1.1 Design Objectives and Design Process

One of the most crucial sections of the Hoover Bus design is the electrical signalling scheme for the datapaths. Unlike the traditional short distance, tight tolerance environment where many details of electrical signal propagation can be overlooked, the Hoover bus deals with a rather non-ideal electrical environment. High capacitances,

crosstalk-susceptible signal channels, long distances between transmitters and receivers, and high bandwidth requirements all present significant engineering challenges to designing the datapaths.

The first step in designing the electrical specification is to examine the requirements and constraints of the system. As stated earlier, the objective is to transmit 80 MB/s between nodes separated by distances up to 6 meters. Since the Hoover bus uses point-to-point links instead of a parallel topology bus structure, the signalling protocol will not have to ensure signal integrity beyond six meters. However, cost and construction constraints prevent the use of exotic technologies such as GaAs or fiber for the initial version of the Hoover Bus.

3.1.1.2 Helpful Techniques

The next step is to examine various theoretical signal integrity improving techniques and to see if any existing signalling protocols are capable of handling the Hoover bus requirements.

3.1.1.3 Differential Signalling

Differential signalling greatly improves signal discrimination in a noisy environment by using the *difference* in voltages across two separate signalling wires, not the magnitude of a single-ended signal against a common reference, to send information. Since both signalling wires are subjected to the same noise sources during transmission, the voltage difference between them is preserved, and the receiver will be more likely to successfully distinguish the transmitted information. The cost of this approach, however, is an increase in the number of signalling wires¹ and an increased component count due to the

higher pin count required to connect the additional signalling wires.

3.1.1.4 Lower Signal Amplitudes and Limited Slew-Rate Signals

Lower signal amplitudes and limited slew-rate signals both decrease the amount of crosstalk between adjacent signal channels. It can be shown [4] that for a bus (shown in Figure 3-1) with a bus delay T_L defined as

$$T_L = L\sqrt{L1 \cdot C1} \quad (3 - 1)$$

where L is the length of the bus
 $L1$ is the distributed inductance of the bus
 $C1$ is the distributed capacitance of the bus,

the crosstalk-induced peak voltage amplitudes at the near end of the transmission line, V_{NE} , and at the far end of the transmission line, V_{FE} , are defined as follows:

$$V_{NE} = K_{NE}(2T_L)\left(\frac{V_I}{t_r}\right) \quad \text{for } (t_r > 2T_L) \quad (3 - 2)$$

$$V_{NE} = K_{NE}(V_I) \quad \text{for } (t_r < 2T_L) \quad (3 - 3)$$

$$V_{FE} = K_{FE}(L)\left(\frac{V_I}{t_r}\right) \quad (3 - 4)$$

$$\text{where } K_{NE} = \frac{L(C_C Z + L_C/Z)}{4T_L} \quad (3 - 5)$$

$$\text{and } K_{FE} = \frac{C_C Z - L_C/Z}{2}, \quad (3 - 6)$$

C_C and L_C are the are the distributive capacitive and inductive coupling between the lines, respectively,
 L is the length of the bus, t_r is the rise time of the bus signal, and
 V_I is the input voltage amplitude

-
1. Although one could argue that the ground wires required for return current paths would negate the lower conductor count of single-ended signalling schemes, the author points out that the number of ground wires can be a fraction of the number of signalling wires.

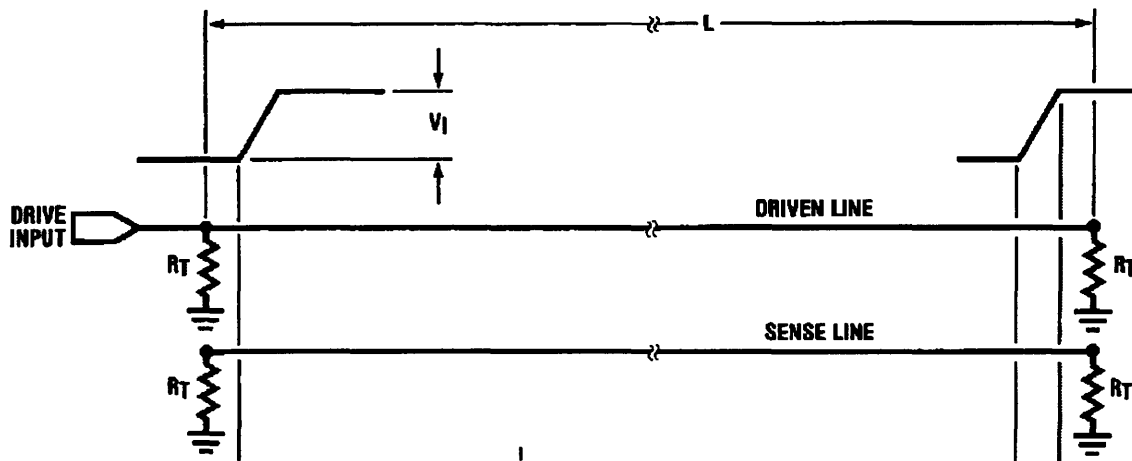


Figure 3-1: Environment for Crosstalk Measurement
(from National Semiconductor Application Note 337)

As one can see, the peak amplitude of the crosstalk voltage at the far end of the conductor (and also at the near end of a short bus) is proportional to the quotient of the input voltage amplitude divided by the rise time of the input signal. Thus, by reducing signal amplitude and using slower transitioning signals, one can reduce crosstalk and raise signal integrity.

3.1.1.5 Termination of Signals

Proper termination of signalling wires reduces signal degradation due to ringing and reflection. For a voltage V_I propagating down a transmission line terminated by a resistor R_T connected to GND, the reflected voltage will have the amplitude $\Gamma(V_I)$ where Γ is computed from the following equation:

$$\Gamma = \left(\frac{R_T - Z_o}{R_T + Z_o} \right) \quad (3 - 7)$$

As one can see, a lack of a termination resistor ($R_T = \infty\Omega$) causes Γ to be 1, and the reflected voltage to be V_I ; conversely, a short to ground ($R_T = 0\Omega$) causes Γ to be

-1 , and the reflected voltage to be $-V_I$. Optimally, one would like to have a termination resistor of resistance Z_o to yield a Γ value of zero and reflected voltage amplitude of 0 volts. However, the elimination of signal reflections is not without cost; there is a current consumption of V_I/Z_o when the voltage is high, so the average power P_T dissipated for a 50% duty cycle signal is $k \left(\frac{(V_I)^2}{Z_o} \right)$, where k is the duty cycle of the signal.

Through derivations examined in Appendix B, one finds that a less energy consuming termination method is active termination, which ties a termination resistor of resistance Z_o from a voltage source of output $(V_I)/2$ to the signal line and has an average power consumption of $\frac{1}{4} \left(\frac{(V_I)^2}{Z_o} \right)$, independent of signal duty cycle.

3.1.1.6 Synchronous Data Transfer

Another useful technique to improve transfer rates is to use synchronous data transfer. In asynchronous bus transfers, the source node sends data with a strobe signal which indicates that valid data is present on the bus. The receiving node latches the data during this period and sends back an acknowledge pulse, indicating correct reception of the data. Although this method allows nodes of differing speeds and capabilities to communicate, it can limit the data transmission speed for long communication channels. For example, during an asynchronous transfer along an eight meter SCSI chain, the target node must wait at least 84 ns before it can change the data lines from their previous value due to the round trip time of the REQ/ACK signals¹.

One of the methods to get around this delay is to not use a full handshake for each byte. Instead, before the actual transfer, the nodes prearrange a data transfer rate that both

1. Assuming a 5.25 ns delay per meter on the SCSI cable and negligible delays on the connectors and inside the nodes.

the transmitter and receiver can handle. Then, during the transfer, the source node can send the data and strobe signal at the specified rate without waiting for an explicit ACK for each byte. In some buses, the synchronous transfer protocols specify that the target node send back an ACK for each byte it received as a flow control mechanism. In such buses, the transmitter and receiver pre-determine the maximum number of data bytes (and REQ pulses) that the transmitter can send ahead of the ack pulses it has received. This way, the receiver unit still has some method of regulating the data transfer. Furthermore, after the transaction has completed, the transmitter can compare the number of ACKs received to the number of data strobe assertions as an error detection mechanism. If the two quantities do not match, an error has occurred, and appropriate error-correction steps can be taken.

3.1.1.7 Backplane Transceiver Logic

After examining these signal improving techniques, Backplane Transceiver Logic (BTL) was examined as a possible electrical signalling protocol for the Hoover bus. The standard had several features which made it especially attractive. First, since it was designed as a bus driving logic family, BTL uses small amplitude, slower edged rate signals to reduce crosstalk problems along transmission lines. Second, its receivers have a low pass filter and pulse rejection circuitry which reject all spurious glitches and transitions which do not conform to properly driven waveforms. Third, the logic family specifies the use of active termination to reduce reflections and reflection induced error.

Although BTL is a single-ended signalling scheme, further examination showed that the costs of differential signalling actually outweighed the benefits. Although noise would be reduced with a differential signalling scheme, doubling the number of signalling conductors would have meant a larger number of bus transceivers at each node. This larger

number of packages would have an increased cost impact upon each slave node, as well as an impact on the board area required for each port on each slave node. Since BTL had a sufficient number of other mechanisms to increase signal integrity, it was chosen instead as the electrical layer for the Hoover Bus.

The data lines are synchronized with HV_CLK – a 40 MHz, NRZ data clock signal (a 20 MHz clock signal whose rising and falling edges correspond to valid clock edges). Since the Hoover Bus uses a packet level REQ – ACK handshaking scheme, and since packets are of fixed, known length, the use of a byte level REQ/ACK synchronous handshaking protocol was deemed unnecessary. Instead, a downstream node is guaranteed to correctly receive and relay data if it reclocks the incoming data upon the rising and falling edges of the HV_CLK signal.

3.1.2 Bus Signal Definition

The bus signals for the Hoover Bus are defined as follows:

Table 2: Hoover Bus Signals

Name	Description	Pin	Pin	Description	Name
D15	Data signal	46	29	Data Signal	D3
D14	Data Signal	45	28	Data Signal	D2
D13	Data Signal	44	27	Data Signal	D1
D12	Data Signal	43	26	Data Signal	D0
D11	Data Signal	42	37	Data clock	HV_CLK
D10	Data Signal	41	34	Arbitration ACK	ACK
D9	Data Signal	40	35	Arbitration REQ	REQ
D8	Data Signal	39	47	Serial+ from Master	HV_TxD+
D7	Data Signal	33	48	Serial- from Master	HV_TxD-
D6	Data Signal	32	49	Serial+ from Slaves	HV_RxD+
D5	Data Signal	31	50	Serial- from Slaves	HV_RxD-
D4	Data Signal	30	13, 14, 15, 16, 17	Ground	GND
GND	Ground	1, 2, 3, 4, 5, 6, 7	18, 19, 20, 21, 22	Ground	GND
GND	Ground	8, 9, 10, 11, 12	23, 24, 25, 36	Ground	GND

Pin assignments refer the standard pin numbering for Centronics SCSI-I and SCSI-2 connectors as defined by the ANSI SCSI-2 Specification X3.131-199x.

3.1.3 Data Packet Definition

The data packet is 514 bytes, with two header bytes and 512 payload bytes. Since the data bus is 16 bits wide, the packet is arranged as 257 short words as shown below.

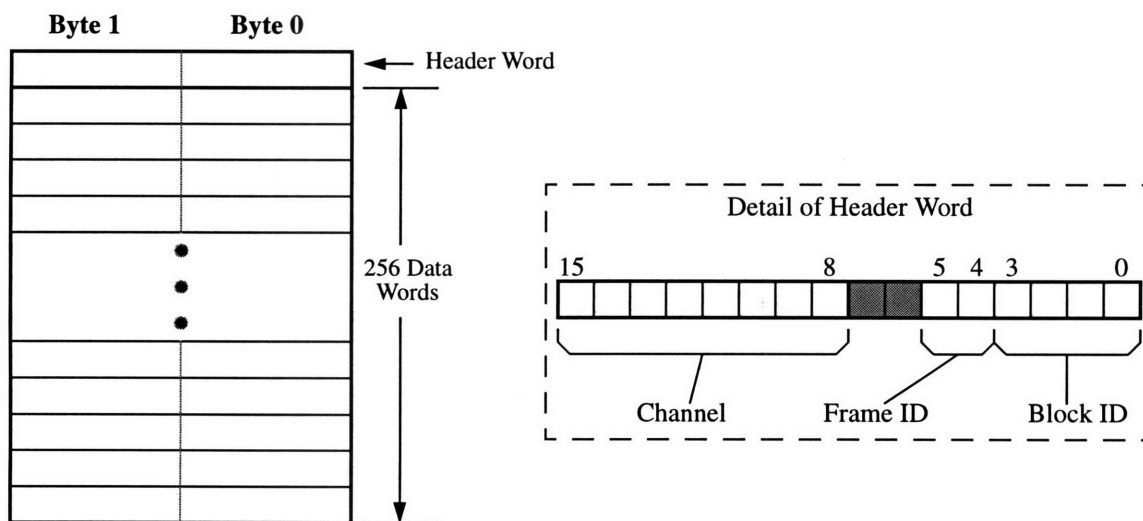


Figure 3-2: Structure of a Data Packet

Bits [15...8] of the header word are defined as the channel ID. Each virtual channel of data has a unique ID; thus, a slave node with three video outputs and two audio outputs would use five separate unique channel IDs.

Bits [3...1] are used as a 16 value block counter to identify the current data packet's position relative to other packets in the stream. Thus, if the slave node fails to transmit a small number (less than 16) of packets, the bus master will be able to detect the error and place the next transmitted packet at the correct location in the memory store. Although there are aliasing problems which prevent the bus master from determining the correct

number of lost packets if more than 16 packets are dropped, the loss of such a large amount of data would render the current frame useless, even if the packets were placed in their correct position. Therefore, the aliasing problems are not a major detriment to the system.

Bits [5...4] are similarly used as a four value frame ID to enable bus master to identify the frame which corresponds to the data packet. Thus, if long-term overflow condition causes a large number of packets to be lost, the bus master can place the next incoming data packets in the correct frame buffer (although not necessarily at the correct position within the frame buffer), and be able to restore correct frame sync once the frame counter increments again. Although the bus master will experience aliasing problems if over four frames of data are lost, such overflow conditions are indicative of much larger problems which will require software to correct.

3.1.3.1 Packet Size Design

Packet size for the Hoover Bus is not an arbitrary choice. Rather, it is a function of several constraints: address generator speed, arbitration overhead, data clock speed, and the likelihood of FIFO overrun likelihood.

The first step in choosing a packet size is to limit the possible packet payload sizes to be powers of two. This limitation is chosen to simplify the address generator programming and possibly boost address generator speed. If a packet size of 2^n is chosen, one needs only to calculate $(m - n)$ address bits to interface to a 2^m bit memory store. This becomes especially important if a memory store larger than 4 GB is used, since any programming steps that involve more than 32 bits will take at least two separate instructions

to implement on most microprocessors.

The next step is to examine the system effects of larger and smaller packet sizes. If the packet size is increased, competing slave units will be able to obtain a bus slot less frequently. Although they will be able to transfer more data once they do obtain a bus slot, the likelihood of slave FIFO overflow is increased unless the amount of slave FIFO is increased accordingly.¹

Small packet sizes have two disadvantages. First, breaking streams of data into smaller packets yields a higher packet interarrival rate. As a result, the address generator gets less time to generate a memory store address for each packet before the next packet arrives. Eventually, as packets are made too small, the address generator will be unable to cope.

Second, since arbitration delay is independent to packet size, smaller packet sizes lead to higher proportions of total bus time devoted to arbitration, as is shown in Table 3.

Table 3: Percentage of Bandwidth Devoted to Arbitration

Packet Size	Transfer time	Typical Average Handshaking Overhead	Wasted Bandwidth	Remaining Bandwidth (in millions of bytes)	Worst Case Average Handshaking Overhead	Wasted Bandwidth	Remaining Bandwidth
256 bytes	3.2 us	1094 ns	25.5%	59.62	1786 ns	35.8%	51.34
512 bytes	6.4us	1094 ns	14.6%	68.32	1786 ns	21.8%	62.55
1024 bytes	12.8 us	1094 ns	7.9%	73.70	1786 ns	12.2%	70.20
2048 bytes	25.6 us	1094 ns	4.1%	76.72	1786 ns	6.5%	74.78

1. This issue was extremely important during the time of the delineation of the bus specification, as the speed of current generation FIFOs dropped below 40 MHz after a capacity of approximately 2 kilobytes was reached.

3.1.4 Arbitration Scheme

One key component which distinguishes the Hoover bus from other existing buses is its arbitration scheme.

Fair arbitration – a strategy which grants bus slots in a manner which minimizes **average** latency for all slave nodes – is the most widely used strategy for general-purpose buses handling non-time critical data. As bus traffic increases, the arbiter tries to ensure that all slave units see an equally increased access time for bus slots. As bus demand exceeds total bus capacity, all the slave units experience internal backlogs as data is produced by the slaves faster than they can deposit it onto the bus. Memory buffers within each slave store the excess packets and deposit them onto the bus as the slave acquires a bus slot. If the demand continues to exceed the total bus bandwidth for enough time, the internal buffers fill up, and internal flow control mechanisms block data production until the bus traffic decreases and the buffers are emptied out.

However, the real-time nature of multimedia data results in different design requirements. Data is produced at regular intervals by mechanisms such as video digitizers and must be sent to a receiving mechanism in a timely fashion. If the receiving mechanism runs out of data to process due to bus congestion or other delays, the result will be a perceptible artifact such as a garbage-filled region on a video buffer frame or a discontinuity in the audio stream; thus, bus requests must be serviced promptly.

Although fair bus protocols can be used for multimedia applications as long as bus traffic is light enough to keep the total slave-to-master transfer time (time for slave to obtain a bus slot + actual data transfer time) less than the packet interarrival time, once bus traffic becomes heavy enough to push the total transfer time above the packet interarrival

time, the fairness in the bus protocol will ensure that all streams will be corrupted. Furthermore, even if the total transfer time on a fully saturated bus is less than the packet interarrival time, FIFO overflow and packet loss remain a problem, since real-time data producing mechanisms such as video cameras cannot be stopped by flow control mechanisms without data loss.

Therefore, the Hoover bus utilizes a fixed, biased arbitration strategy which gives higher priority slaves a better chance to deposit their packets onto the bus, ensuring that as bus demand is increased beyond total bus capacity, a maximum number of slaves will be able to continue to send their streams uncorrupted.

Slaves desiring a bus slot assert their downstream REQ line and wait for an ACK reply from the bus master before initiating a bus transfer. After the slave finishes its transaction, it lowers the REQ line, and shortly afterwards, the ACK line is lowered when the bus master sees the REQ line deasserted.

When one or more slaves lie between the requesting slave and the bus master, each slave has the choice of either allowing the original request to propagate downstream towards the bus master, or if the slave has a packet that it wants to send, blocking the request and initiating its own request. There are several variations of this algorithm which can be implemented in the arbiter FSMs, resulting in differing behaviors when collisions occur; the primary one will be discussed in detail below, and each of the variations will be described briefly afterward.

3.1.4.1 Slave Arbitration FSM - Primary (“Greedy”) Model

The arbitration finite state machine within a slave node is broken down into three

separate, smaller FSMs to simplify the logic required for implementation and to increase the maximum operating speed of the arbiter. As one can see from the diagram below, each state machine is responsible for one of the output signals in the arbiter:

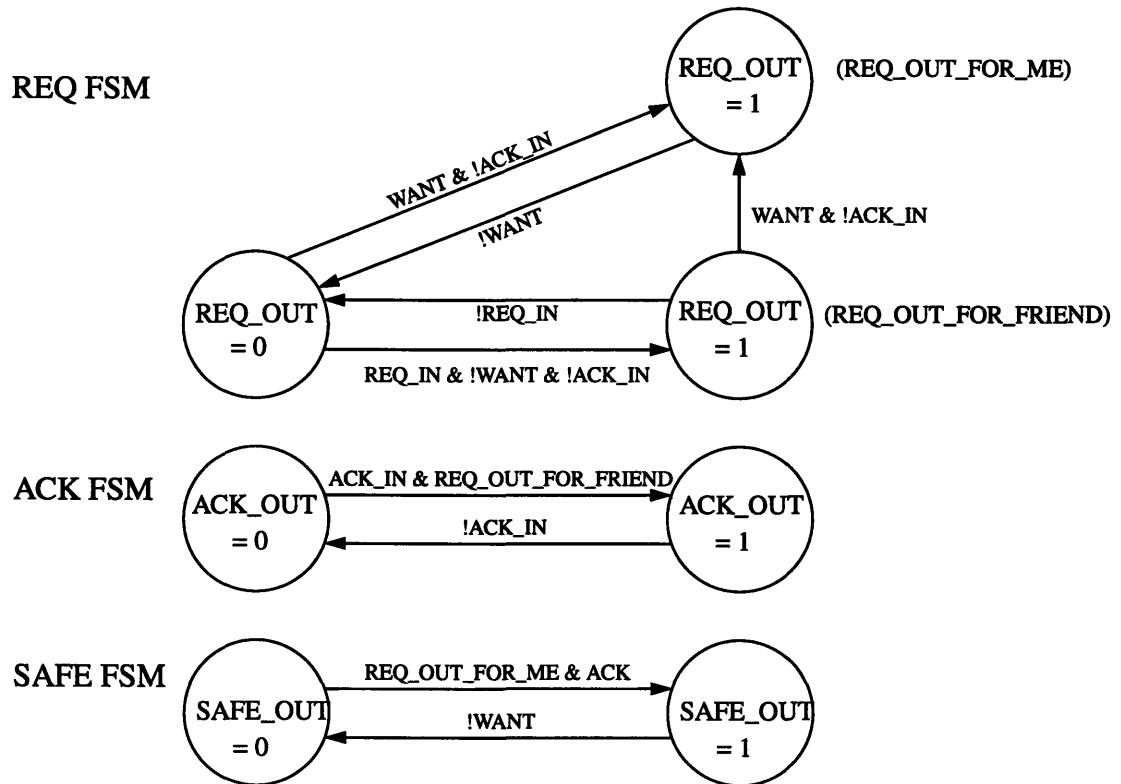


Figure 3-3: Primary State Machine for Arbiter

In order to best understand these state machines, one should examine the machines as they are exposed to all possible sets of inputs as they are in the idle state, which is indicated by (REQ_OUT = 0), (ACK_OUT = 0), and (SAFE_OUT = 0). The table below shows all possible sets of inputs and the procedures executed by the FSMs in response to the inputs. Note how the FSMs only take action to sets of inputs which occur during nor-

mal operation of the bus; all erroneous cases are ignored.

Table 4: All Input Combinations to Slave Arbiter FSM

INPUT SIGNALS			ACTION TAKEN
ACK_IN	REQ_IN	WANT	
OFF	OFF	OFF	NONE
OFF	OFF	ON	CASE 1
OFF	ON	OFF	CASE 2
OFF	ON	ON	CASE 1
ON	OFF	OFF	NONE
ON	OFF	ON	NONE
ON	ON	OFF	NONE
ON	ON	ON	NONE

CASE 1: If the node is idle, the consumer wants a bus slot (WANT is high), and the node has not received an ACK yet (meaning that the node is truly idle, and not just deasserting REQ_OUT to end a transaction), the arbiter goes to REQ_OUT_FOR_ME, asserts REQ_OUT and continues to stay in this state until WANT is deasserted by the consumer, indicating that the consumer wants to end a transaction.

While REQ_OUT is asserted, the ACK FSM must stay in (ACK_OUT = 0). The SAFE FSM stays deasserted until the ACK signal has been received from the bus master, indicating an bus access slot for the node. At this time, the SAFE signal is asserted, and data transfer is initiated.

While the data transfer happens, nothing can change the state of the node except for the deassertion of the WANT signal. Once this occurs, data transfer is done, and the SAFE signal is deasserted. Also at this time, the REQ_FSM returns from

REQ_OUT_FOR_ME to (REQ_OUT = 0), and the slave node returns to the idle state.

CASE2: If the node is IDLE and the REQ_IN signal is asserted on its upstream port, another slave wants the node to relay its request for a bus slot. If the consumer simultaneously wants access to the bus (that is, the WANT signal is asserted at the same time), it will win since this node has higher priority over its upstream neighbors. If the ACK signal is currently asserted, this node is presently servicing its own bus access slot, and the arbiter ignores the upstream neighbor. However, if neither of these conditions is true, it is safe to service the upstream neighbor's request, and the REQ FSM goes into the REQ_OUT_FOR_FRIEND state.

While the slave waits for the ACK_IN reply from the bus master to relay to the upstream segment, the arbiter can still honor requests from the consumer by moving the REQ FSM from REQ_OUT_FOR_FRIEND to REQ_OUT_FOR_ME. At this point, the slave is in the state described in the previous case, and the consumer's request is serviced in the appropriate manner.

However, if the arbiter receives the ACK_IN signal before the consumer asserts WANT, the bus master has cleared the upstream slave's request, and node N simply relays the granted slot by moving the ACK FSM into the (ACK_OUT = 1) state. At this point, the REQ FSM is forced to stay in the REQ_OUT_FOR_FRIEND state, the SAFE_FSM is forced to stay at the (SAFE_OUT = 0) state, and the node simply acts as a relay until the REQ_IN signal is deasserted. Once this occurs, the REQ FSM returns to the (REQ_OUT = 0) state, and the ACK FSM returns to the (ACK_OUT = 0) state when the bus master

completes the closing of the transaction with the deassertion of the ACK_IN signal (note that the REQ FSM must stay idle until the node returns to the idle state. Any activity by the consumer or upstream neighbor is ignored until the node is fully idle).

3.1.4.2 Slave Arbitration FSM - “Generous” Model

A “generous” variation of the primary model was developed as an alternate test model. As one can see in Figure 3-4, the state machine is nearly identical to the previous FSM. However, the one missing state transition, the transition from REQ_OUT_FOR_FRIEND to REQ_OUT_FOR_ME, is quite significant. Unlike the greedier primary model which intercepted ACKs intended for upstream neighbors when the consumer desired bus access, the generous model allows these ACKs to pass upstream to the requesting node. Thus, this FSM sacrifices part of its priority to yield a slightly more fair bus.

If the bus implementer inserts an external control signal into the arbiter FSM, the user will be able to choose between different versions of the arbiter, and may be able to experiment to find which variant yields the best behavior for his or her particular bus con-

figuration.

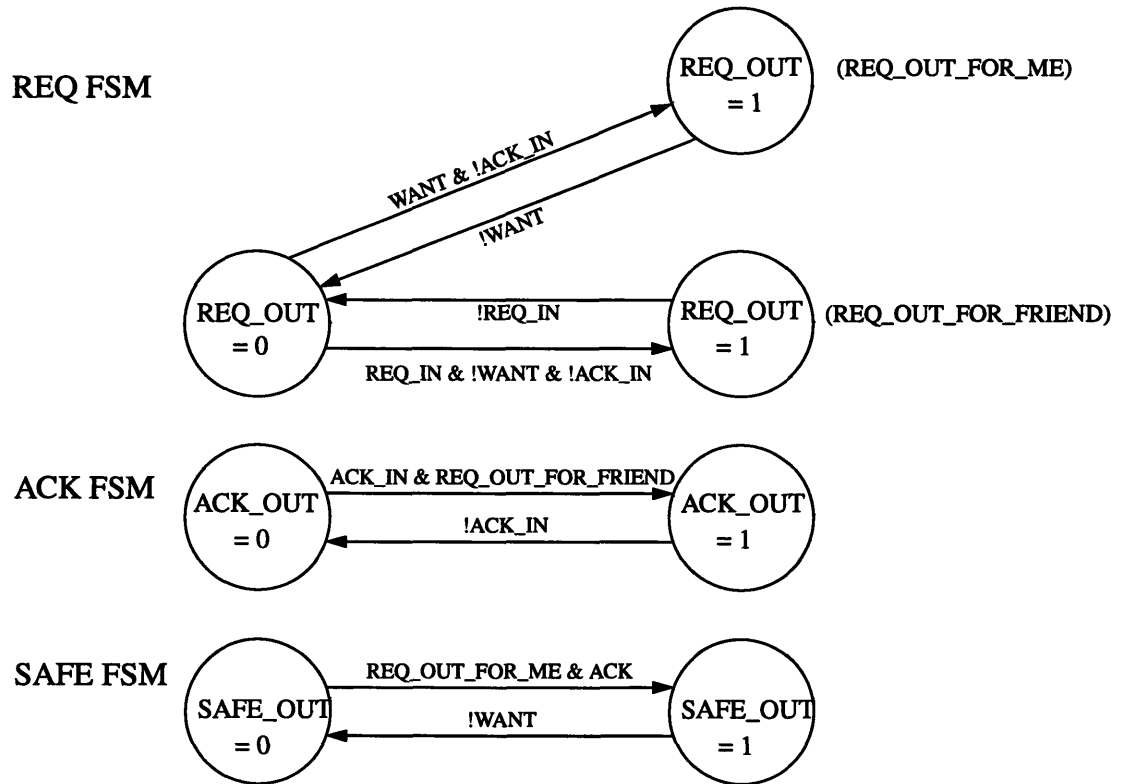


Figure 3-4: Generous Arbiter FSM

3.1.4.3 Other Variations of the Arbiter FSM

Several other variations of the arbiter FSMs have been suggested during the design phase of the Hoover Bus but have not been implemented to date.

One proposed change to the generous arbiter FSM is to have a counter in the arbiter which gives a finite time limit to an upstream node to request a bus slot from the bus master. This change would shorten a node's wait for the bus if there is a significant delay induced by heavy traffic in the downstream segment of a bus. The rationale behind this alteration is that although a node can temporarily give up the bus for an upstream neighbor, it cannot risk giving up the bus to its neighbor for extended periods of time which

might cause data loss. In such a case, the node would revert to a greedy algorithm that would expedite its own requests.

3.1.4.4 Examples of Hoover Bus Timing

Several timing diagrams are shown below describing the operation of the Hoover bus under several conditions.

3.1.4.5 Simple Hoover Bus Timing

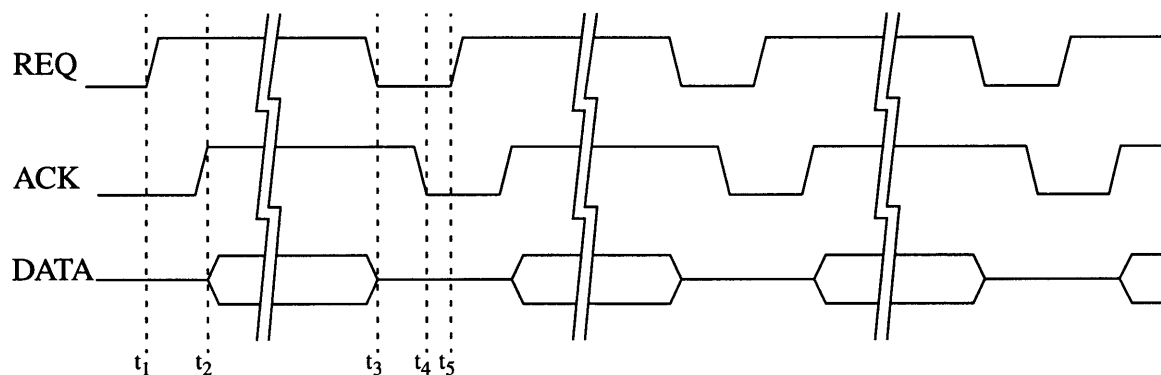


Figure 3-5: Overview of Simple Hoover Bus Timing.

In the above timing diagram, the arbitration signals and data signals are shown for a single slave transferring packets at maximum speed without competition from neighboring nodes. At time t_1 , the slave asserts its REQ signal, requesting a bus slot. At time t_2 , the bus master's acknowledgment has travelled back to the slave unit, and the slave begins transferring data. At time t_3 , after all packet data has been sent, the slave deasserts the REQ line, and shortly afterwards, at time t_4 , the bus master deasserts its ACK line. After a short delay used to clean up internal state machines, the slave unit begins repeating the cycle at time t_5 .

3.1.4.6 Greedy Slave Arbitration Timing

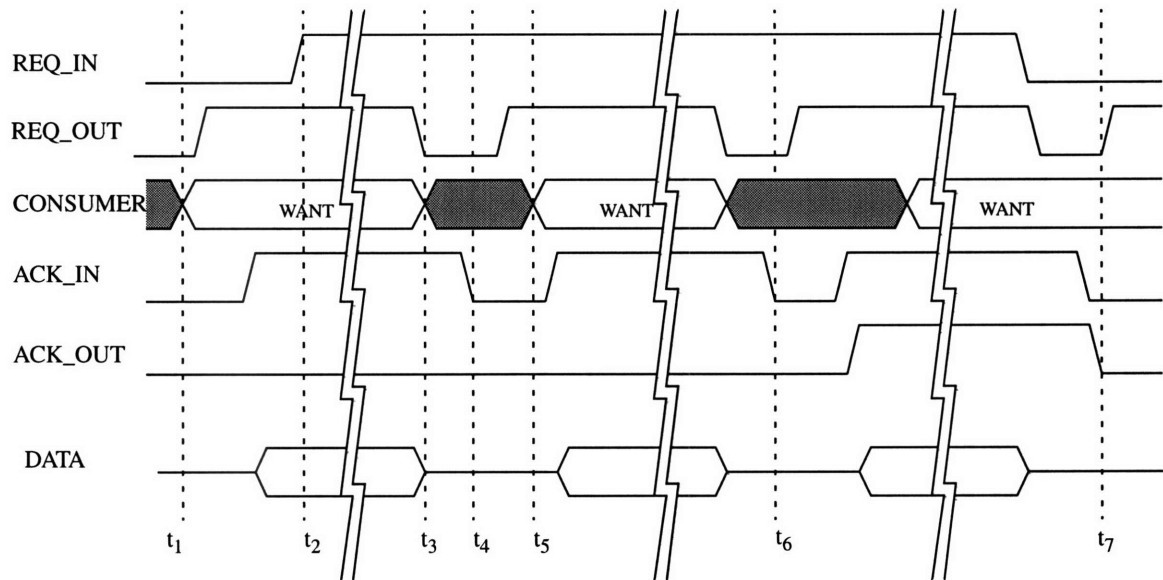


Figure 3-6: Greedy Slave Arbitration Timing Diagram

In a situation where there is competition for bus slots, it is necessary to look at the state of a slave node's internal consumer and the signals present on its upstream and downstream ports. In the timing diagram above, a slave node, N, using the "greedy" variant of the arbiter FSM decides to send data at time t_1 . It proceeds in the same fashion as the slave in the previous diagram did, asserting its REQ signal and initiating a data transfer when it receives the ACK assertion on its downstream port.

At time t_2 , N's upstream neighbor, node M, requests a bus slot by asserting the REQ_IN input on this node's upstream port. However, it is not until time t_4 , at which point node N has finished transferring data and has received the deasserted ACK signal from the bus master, when node N actually begins servicing M's request by reasserting its REQ line again.

At time t_5 , N's consumer FSM decides that enough data exists in the FIFO buffer to request a bus slot. Since the bus master's ACK to M's request has not been sent yet,

node N's greedy algorithm decides to intercept this ACK and use it to satisfy its own internal REQ instead of passing it upstream to node M.

After servicing its own bus slot and deasserting its REQ_OUT line, node N must again wait until time t_6 for the bus master to deassert the downstream ACK input before reasserting REQ_OUT again for node M's request. This time, the ACK from the bus master is received and relayed to node M before N's producer again wants to use a bus slot. As a result, N's arbiter ignores its own consumer and allows node M's request to finish before arbitrating for its own bus slot at t_7 .

3.1.4.7 Generous Slave Arbitration Timing

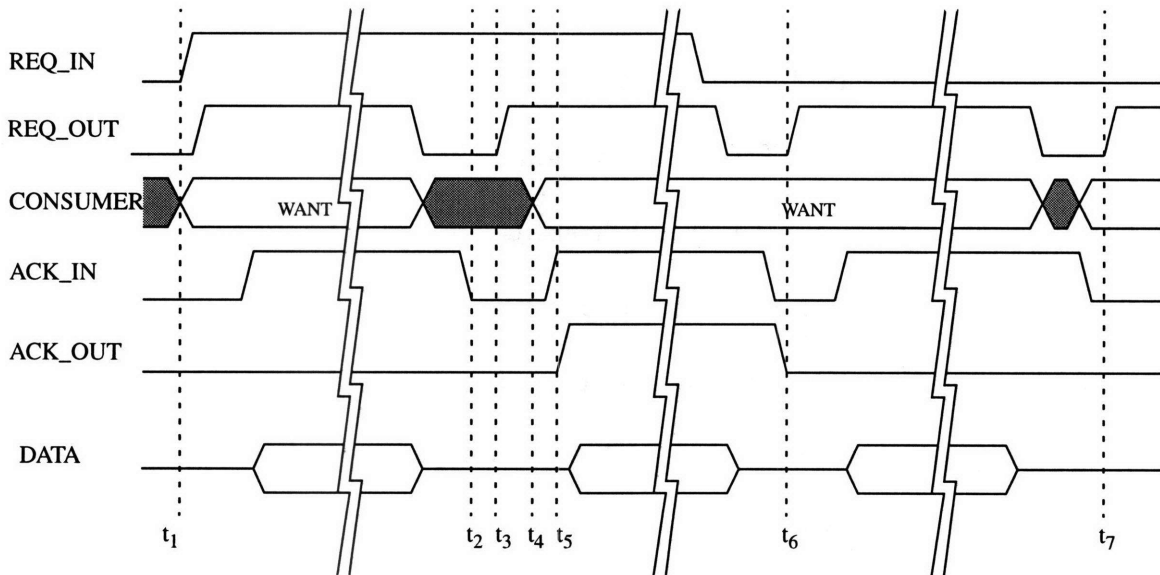


Figure 3-7: Generous Arbitration Timing

In this example, N's slave arbiter is replaced by the generous variation. The first bus transaction (from t_1 to t_2) occurs in the same manner as before. As expected, at time t_3 , node N asserts the REQ_OUT line to relay a request for node M's previously asserted REQ_IN input.

However, at time t_4 , when node N's internal consumer detects valid data for a bus request, the generous arbiter does not allow the consumer to intercept the upstream neighbor's request for bus access. Thus, at time t_5 , the arbiter relays the bus master's acknowledgment to the upstream bus segment, and waits until time t_6 when this bus slot has completed, before attempting to service this node's request for bus access.

Note that because of the delay in servicing N's request, an additional packet has accumulated behind the packet being serviced at time t_6 . Thus, immediately after the packet is serviced, the WANT signal is reasserted by the consumer, and the arbiter begins servicing N's backlogged data.

3.2 Command Protocol

The serial protocol used for bus master to slave communication was designed to allow fast, simple message passing in a multitasking environment while adhering to the limitations of the hardware chosen in the construction of the Hoover bus on the I1 Card in the Cheops system. The expected traffic patterns were simple, small (two character opcodes with options for arguments, if necessary) commands from the bus master followed by reply packets from the selected slave with either a simple ACK or return values from executing the master's command.

3.2.1 Electrical Layer

The serial network is a RS-485 four wire multidrop network consisting of one differential pair of wires which connects the bus transmitter to all the slave receivers and another differential pair which connects the slave transmitters to the master receiver (see

Figure 3-8).

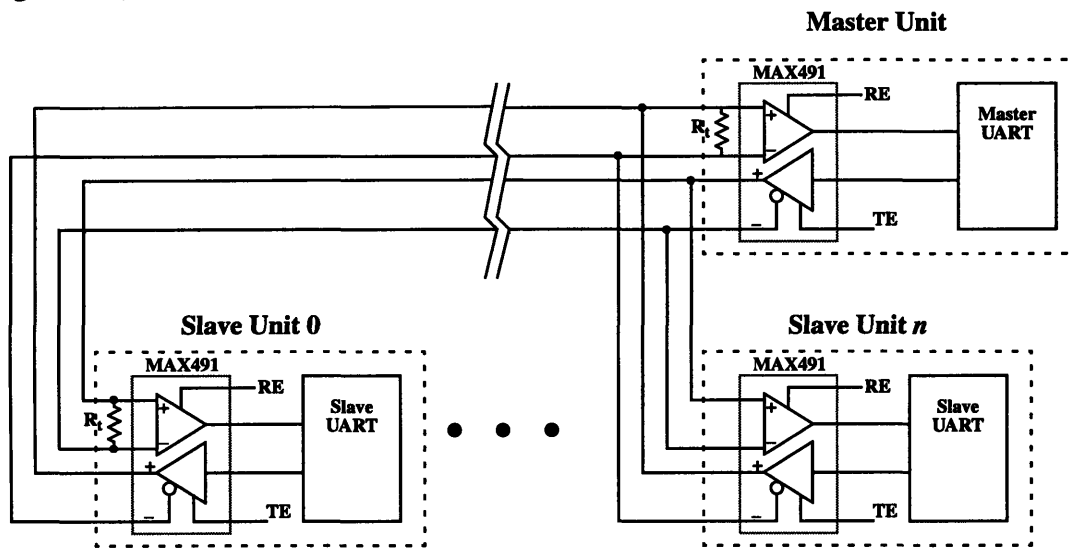


Figure 3-8: RS-485 Four Wire Multidrop Network

Although it was possible to save two wires by using only one differential channel for all the transmitter and receiver elements, the four wire multidrop configuration had several advantages. First, by isolating the master to slave bus and slave to master bus, we can eliminate packet ambiguity. The slaves transmitters can only connect to the master receiver, so slave receivers do not have to parse other slave to master packets. Furthermore, high efficiency protocols such transaction pipelining and split phase transactions become much more difficult to implement with a common serial channel, since the chances of contention between the nodes becomes much higher in such a structure.

RS-485 was chosen as the electrical standard for the serial channel because it fulfilled the requirements of our application. First, its differential signalling scheme makes it more resistant to noise and crosstalk -- a serious problem since the serial bus wires are in the same bundle as the datapath signalling wires. Second, RS-485 is designed for long distance networks (1200 m) with more than one transmitter and receiver on the bus. Third, the theoretical maximum data rate of 10 Mbits/second greatly exceeds the expected com-

mand rate.

3.2.2 Protocol Design Considerations

The serial communications protocol design was heavily affected by the choice of the devices on both ends of the serial channel. The MC68HC11 MCU used in the slave devices has a unique receiver wakeup feature which simplifies protocol design in situations where one transmitter selectively sends messages to one or more receivers sharing a common serial channel. The receivers monitor the common channel for the start of a message from the transmitter. Upon detecting this starting condition, all the receivers wake up, process the first byte of the message, and determine if the message applies to them. Those not addressed by the transmitter shut down their receivers until another message is sent out, and the rest keep their receivers active for the duration of the message.

The 68HC11 has two methods of implementing this wakeup feature. The first variant, called idle-line wakeup, uses line conditions to determine the message boundaries. When the serial line has been idle for one character's worth of time, all the receivers assume that a message has ended; the next transmitted byte will be the start of a new message which should be processed to determine whether or not the message applies to them.

The second variant, called address-mark wakeup, uses the most significant bit (MSB) of each byte to determine message boundaries. If the MSB is 1, the receivers know that this byte is the first byte of a message, and process it to determine if the message is addressed for them. The receivers then treat all bytes with a MSB of 0 as part of the current message. The next message is again indicated by a byte whose MSB is one.

A third alternative is to use a message format that does not rely on either timing

constraints or character set constraints to establish message boundaries. Rather, the master sets the slaves to a known state via a unique signalling mechanism (such as a large sync packet or a break) and then begins serial communication. The slaves listen to the serial stream and maintain sync by parsing each message from the master. Since each message begins with a header that can identify the packet type and size, the slaves can simply count the bytes to determine the next message boundary.

Each of the above methods has limitations which have to be considered in the context of this project. Initially, the third approach seemed to be the most flexible and desirable solution since it did not have the limitations of the other schemes; however, it was also the least robust of the options. Periodic sync packets would have to be interjected into the channel to resynchronize slave units which lost sync through buffer overflow or framing errors. In addition, the inclusion of additional data such as opcode and size bytes would add sufficient overhead to packets which were initially envisioned to be two bytes or less in size.

Although the address-mark wakeup mode has the limitation of forcing the MSB's of all the message bytes to zero (and thus making the task of encoding numerical arguments for a serial command more difficult), the idle-line wakeup mode's timing constraints present a more serious problem to a cooperative multitasking environment like the Magic 7 OS found in Cheops. A poorly written application may cause a message-sending process to cease serial transmission temporarily, causing the slaves to interpret the next byte in the message as the beginning of another message. Furthermore, since few commands in the expected master-slave message set utilized more than one character of opcode in the body of the message, the loss of throughput due to encoding numeric argu-

ments in 7 bit format was less than initially expected. Thus, the address-mark wakeup mode was used as the receiver addressing scheme, and necessary adjustments were made to the serial protocol to only use seven of the eight bits per byte¹.

The choice of the MC2681 DUART as the interrupt/serial communications manager limited the baud rate selection that we could easily implement. Of the available set, the fastest rate of 38.4 KBaud was chosen for transmission speed².

3.2.3 Messaging Protocol

Commands on the Hoover Bus consist of single phase, unpipelined bus transactions on the serial bus which are initiated by the master and followed by a reply from the addressed slave unit. Bus commands are used to perform actions such as enabling a slave's digitizers, checking slave error status, changing digitization parameters, and obtaining slave identification information. The protocol is asymmetrical in nature since constraints are different for master-slave command and slave-master replies³. Since the master has a closed set of commands with fixed argument formats, its messages are fixed-length with the message size inherently tied to the opcode. On the other hand, depending on the relative success of executing the command, slaves can return several different replies to a command; thus reply packets are variable length and have a dedicated field which contains the size of the packet. Furthermore, since the MSB considerations necessary for addressing multiple slave units are not necessary for slave to master replies, the reply packets can

-
1. Although the 68HC11 does have the option of choosing nine bits per serial character which would allow the use of eight bits per character for data transmission, the MC2681 DUART does not support this format.
 2. See section 4.2.6 for further details behind this limitation.
 3. See section 3.2.3 for the details behind the constraints.

use straight binary encoding of numerical information, reducing computational stress on the slave MCUs.

3.2.4 Existing Serial Protocol Opcodes

A closed set of serial opcodes developed for the test slave devices discussed in Chapter 4 are shown below for reference. This list of opcodes is expected to grow significantly as more slaves with additional command requirements are constructed.

Table 5: Serial Message Protocol

Master->Slave Opcode	Operands	Meaning	Possible Replies from Slave
'RS'	none	Reset slave	ACK successful
'VS'	none	Version query	STRING version reply string
'BT'	<n> where $0 \leq n \leq 255$	Begin hoover transfer for channel n .	ACK successful ACK error
'ET'	<n> where $0 \leq n \leq 255$	End hoover transfer for channel n .	ACK successful ACK error
'MH'	none	Memory test for header region	ACK successful ACK error <address of fault>
'MD'	none	Memory test for data region	ACK successful ACK error <address of fault>
'PT'	<n> where $0 \leq n \leq 9$	Load pattern test <n>	ACK successful

Table 6: Slave Reply Formats

Opcode	Meaning	Additional Data
0x00	Parse error	none
0x01	Overflow error	none
0x02	Framing error	none
0x10	ACK successful	none
0x11	ACK error	address of fault, error code
0x12	STRING	reply string

3.2.5 Structure of Master to Slave Serial Messages

The structure of the master to slave messages is shown below to clarify the protocol structure. As noted previously, address-mark wakeup mode was chosen for master to slave messages. Thus, only byte 0's MSB is set to one; all following bytes are limited to the ASCII set of characters to encode data.

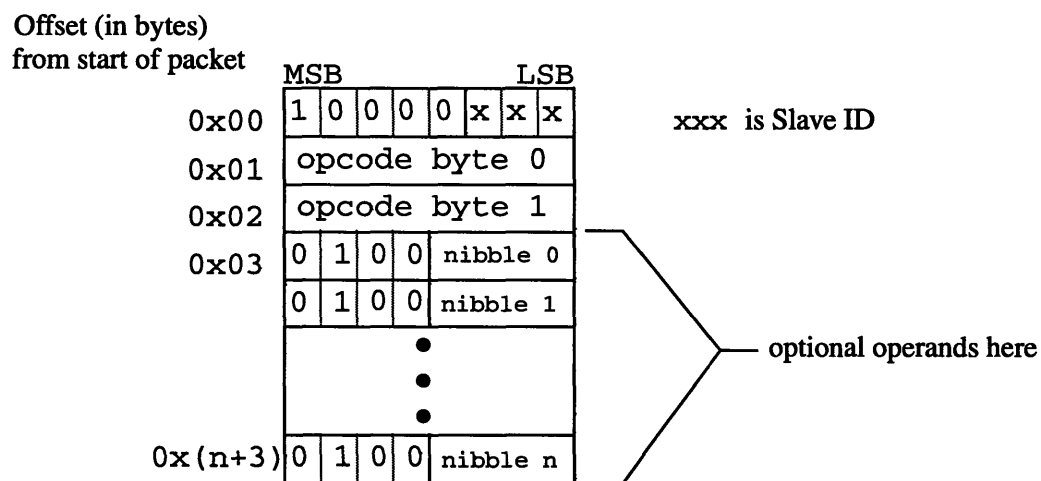


Figure 3-9: Structure of Master to Slave Message

The target slave's address is stored in the lower three bits of byte 0 to allow each slave to determine if the message is directed at them. The next two bytes are the opcode as

defined in Table 5. The remaining bytes are then used to encode numerical arguments.

3.2.5.1 Encoding of Numerical Arguments

Since the serial protocol reserves the MSB of each byte to indicate message boundaries, a method of encoding binary data without using eight bits per byte was needed. The method chosen here is to package each nibble of a numeric operand into each byte of the message. Initially, the simplest solution was considered: to package the nibble into the lower four bits of the message byte and set the remaining bits to zero.

However, the ASCII character set associated with 0x00 to 0x0F consists of a variety of control characters, including NUL, SOH, and STX. Although one can write these values easily in C, a more elegant solution for accessing these characters in the 68HC11 is to turn on the seventh bit of all operand bytes, thus mapping the numeric arguments 0x0 - 0xF to the more easily represented characters '@', 'A', 'B', ..., 'O'.

3.2.6 Structure of Slave to Master Reply Message

The reply messages sent from the slave to the bus master are structured differently since they do not have to obey the MSB considerations of master to slave messages and since they are variable length. As a result, bits [7..3] of byte 0 are used to encode the size of the message. In addition, the two ASCII opcode bytes in the master to slave messages are replaced by a single opcode byte between values 0x00 to 0xFF. The remaining bytes

are then allocated for reply packet data.

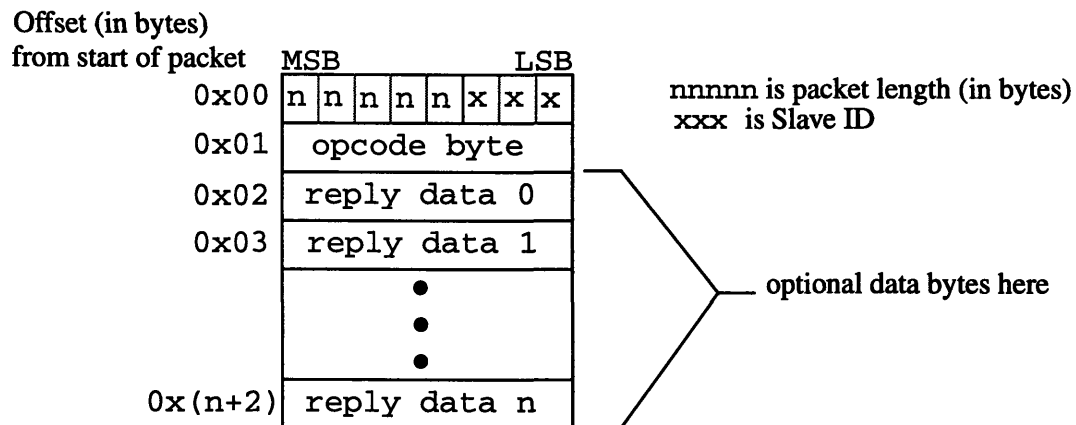


Figure 3-10 Structure of Slave to Master Reply Message

3.2.7 Flow Control

One potentially interesting use for the serial bus communication protocol is automatic flow control through software mechanisms. Suggestions have been made to have a traffic monitoring daemon in the address generator which determines the degree of saturation on the bus by counting missing packets from each active channel. If packet losses on the bus become excessive, a serial transmission daemon can automatically decimate sources in order to reduce traffic on the bus by identifying lower priority channels through a *priority table* which ranks the value of individual channels and by sending decimation commands to these lower priority channels.

3.3 Slave Design Overview

A Hoover Bus slave unit can be broken down into four separate functional blocks

to enhance flexibility of design and simplicity of implementation.

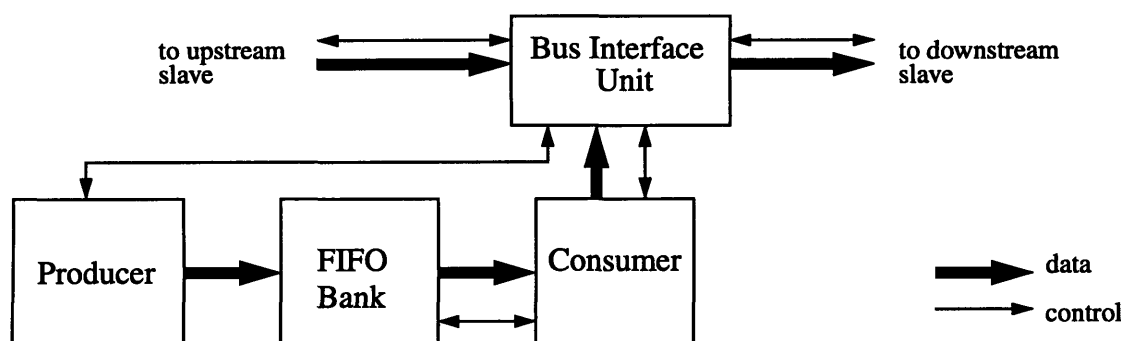


Figure 3-11: Block Diagram of Slave Unit

3.3.1 Data Producer

The **producer** is the data producing mechanism on the slave unit. It can be a device with one channel of output, such as a monochrome video digitizer, or a multichannel device such as an RGB digitizer with several additional audio outputs. Control signals and status flags are sent through an asynchronous register interface in the bus interface unit. Since the producer interfaces to a FIFO buffer with independent read and write clocks, data production rates can be asynchronous with the rest of the slave logic, as long as the average rate of data production is slower than the maximum rate of transfer to the bus master.

3.3.2 FIFO Buffer

The **FIFO buffer** acts as a repository to store data from the producer between the time when the producer initially generates data and the time when the consumer has successfully obtained a bus slot to initiate a Hoover transfer. As data is fed into the FIFOs, status flags signal the amount of data to the consumer FSM, which then decides whether a Hoover transfer should be executed. During a transfer, the FIFO is asynchronously emptied by the consumer while the producer continues to feed data into the input port. If an

overflow condition occurs (the consumer fails to clear enough room from the FIFO for the producer's data), the FIFOs are responsible for a recovery from the failure which results in the loss of an integral number of packets, preserving the synchronicity of the remainder of the stream.

3.3.3 Consumer

The **consumer** is composed of two pieces of circuitry: a *control logic module* and an *output register*. The control logic module requests a bus slot from the arbiter (through the WANT signal) when sufficient data is available for a packet transfer and clocks a single packet's worth of data through the output register once a bus slot has been obtained (indicated by the assertion of the SAFE signal by the arbiter FSM). In a slave whose producer only generates one stream of data, the consumer logic module can be a simple FSM which asserts the WANT signal whenever the half-full flag for the FIFOs is asserted and then asserts the FIFO read enable flags for 256 clock cycles. In a more complicated multi-stream producer case, the consumer will have to use a cycling mechanism to check the FIFOs for each channel, requesting bus slots whenever the presently accessed channel's FIFO has data, and jumping to the next channel after a single packet of data is sent to the output latch (see Section 3.3.6. for more details).

3.3.4 Bus Interface Unit

The **bus interface unit** (BIU) is composed of several functional blocks which together serve three purposes. First, it arbitrates for bus slots when the consumer determines that a packet should be sent to the bus master. Second, when the slave unit is depositing data onto the Hoover Bus, the BIU converts the data signals from TTL levels used for

internal logic into Bus Transceiver Logic (BTL) levels used on the cable segments between the slave nodes. Third, it transfers and buffers data from the upstream bus segment to the downstream bus segment.

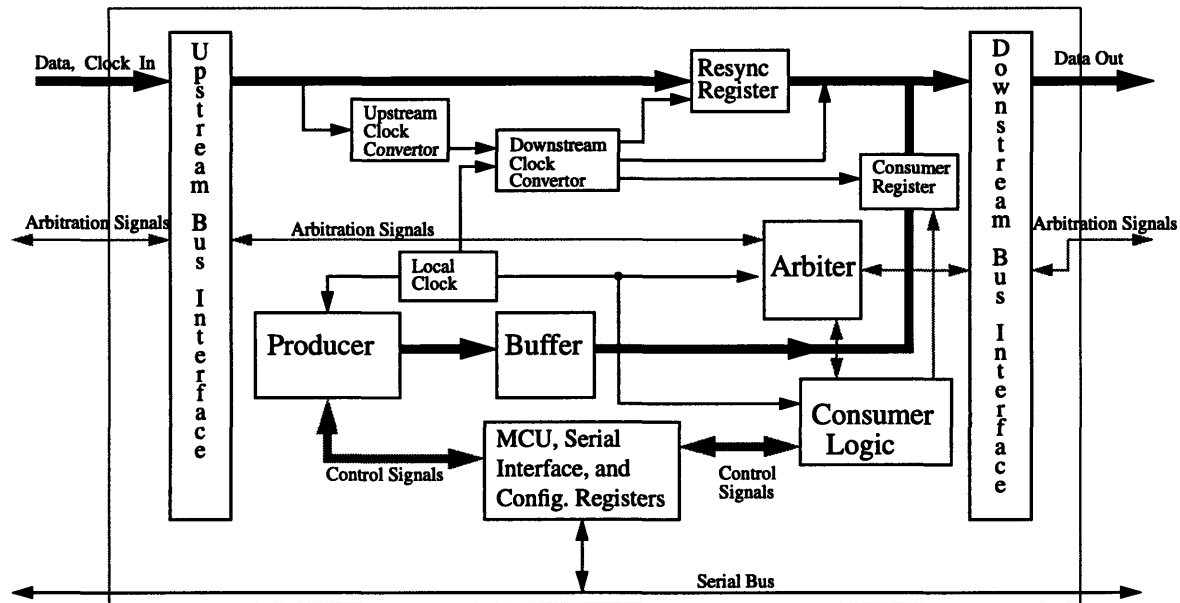


Figure 3-12: Expanded Block Diagram of Slave Unit

The **arbiter** FSM in the bus interface unit has six signals distributed among three ports (see Figure 3-13). The first port, the *downstream port*, connects to the adjacent slave unit closer to the bus master (or to the bus master itself); it contains the slave's request line (REQ_OUT) and the acknowledge input (ACK_IN). The *upstream port* connects to the adjacent slave unit further away from the bus master, or if the slave unit is the furthest upstream node on the bus, the upstream port is left unconnected. Requests from upstream slave units (REQ_IN) and acknowledge signals to these requests (ACK_OUT) are fed through this port. The *consumer port* connects to the producer mechanism inside the slave unit; when the consumer detects that the FIFOs have enough data, it asserts the WANT

input to the arbiter; the arbiter responds with the SAFE output.

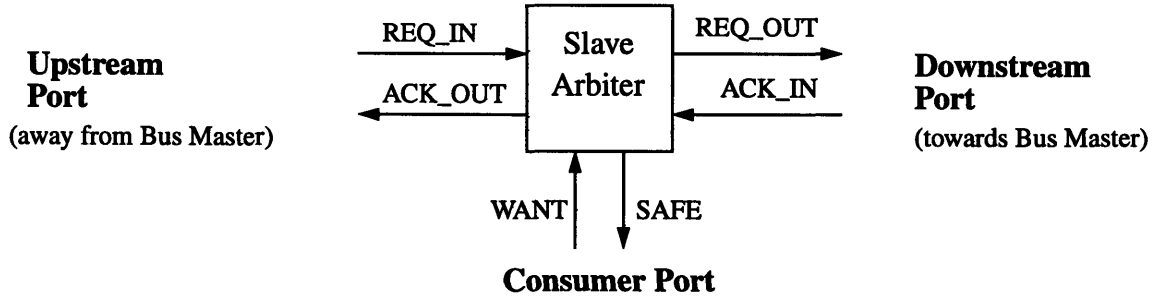


Figure 3-13: Slave Node Arbiter Signals

The **upstream and downstream bus interfaces** translate data and arbitration signals between internal representation (TTL levels) and external representations (BTL levels). The upstream interface receives D[15..0], ACK, and HV_CLK from upstream devices, and relays REQ signals from downstream. Conversely, the downstream interface receives the REQ signal from the connected cable segment and relays retimed copies of D[15..0], ACK, and HV_CLK received from the upstream port. Active termination circuitry consists of 110 ohm resistors (to match the impedance of the SCSI cables) tied between each signal line and an active 2 volt DC source to minimize reflections.

The **upstream clock convertor and downstream clock convertor** maintain data synchronicity on the slave and on the downstream bus segment. The upstream clock convertor provides a clock to resynchronize data transferred in from the upstream port. As the electrical layer described, the HV_CLK is a 40 MHz, NRZ clock signal. The upstream clock convertor produces a conventional 40 MHz, return-to-zero clock signal, 2xHV_CLK, from the bus clock which is then fed to the downstream convertor.

The downstream convertor generates three timing signals from its two inputs: 2xHV_CLK and the local 40 MHz crystal which drives the rest of the slave. The first out-

put is the HV_CLK for the downstream port which is either a divided version of 2xHV_CLK, a converted version of the local clock signal, or GND if no transfers are occurring.

The second output is the clock for the **resync register**, which deskews data from the upstream port in preparation for transmission downstream. If data is not being transferred from the upstream port, this signal is GND; otherwise, this signal is a delayed version of 2xHV_CLK. The delay is introduced to compensate for the propagation delay in the clock divider (see Section for the timing specifications for this subsystem).

The third output is the clock for the producer register. If a Hoover transfer from the internal FIFO is occurring, this signal is a delayed version of the internal 40 MHz clock; otherwise, the signal will be GND.

3.3.5 MCU and Control Logic

The MCU and control logic block for Hoover slaves are used to interpret incoming serial commands from the bus master and execute configuration programs based on these commands. Motorola MC68HC11E2 microcontroller units are suggested as the MCU of choice for this particular component of the bus slave, due to their built in serial interfaces, high versatility and availability on the commercial market. Slave core controller code can be written in Motorola assembly language, and uploaded into internal EEPROM via the built-in serial port.

3.3.6 Producer - FIFO Interface and Overflow Mechanisms

In order to support recoverable overflow errors in the Hoover bus, it is not sufficient to use a single FIFO as the storage medium for a producer. If the FIFO becomes full

in such a system, bytes will be lost for an indeterminate amount of time until the consumer can empty data from the FIFO. All future packets in the stream will have their data words offset by $(x \text{ MOD } 512)$, where x is the number of bytes missed during the overflow period.

3.3.6.1 Overflow Handling with Single Stream Producer Outputs

To ensure that overflows result in an integral number of packets being dropped from the stream, a single packet-sized auxiliary FIFO can be placed at the producer output. Consider a producer mechanism such as a monochrome digitizer which yields a single output stream. An auxiliary 512 byte FIFO is placed between the producer output and the larger, main FIFO memory. If the main FIFO memory store is full when the auxiliary FIFO becomes full, the overflow manager FSM clears the auxiliary FIFO, resulting in the loss of an entire packet.

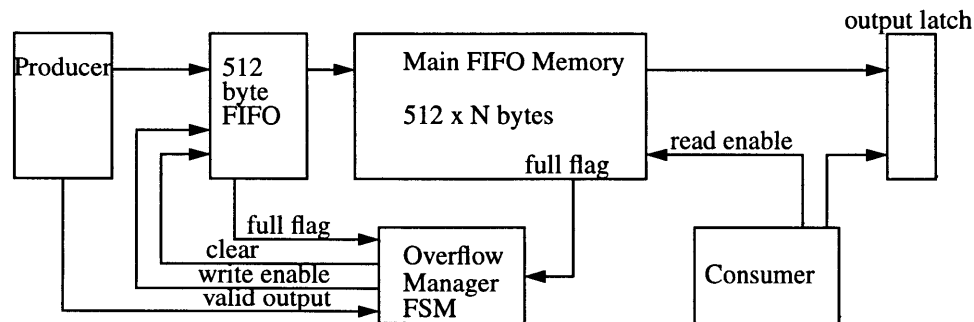


Figure 3-14: Overflow Handling via Auxiliary Pre-FIFO

3.3.6.2 Interfacing the Producer and FIFO - Multiple Symmetric Outputs

If the producer outputs several symmetric streams (such as in the case of a color digitizer with red, green, and blue outputs), a more complicated FIFO structure and control circuitry block are required to make sure the output streams stay synchronized through overflow conditions. Each channel has a separate auxiliary FIFO and main FIFO store, and

the outputs of the main FIFOs are muxed together to the output latch. When an overflow occurs on any one channel, the FIFO overflow manager FSM removes packets from the auxiliary FIFOs on all the producer's output channels, not just the channels with the overflow condition, as this could cause synchronization errors under certain conditions (see Figure 3-15).

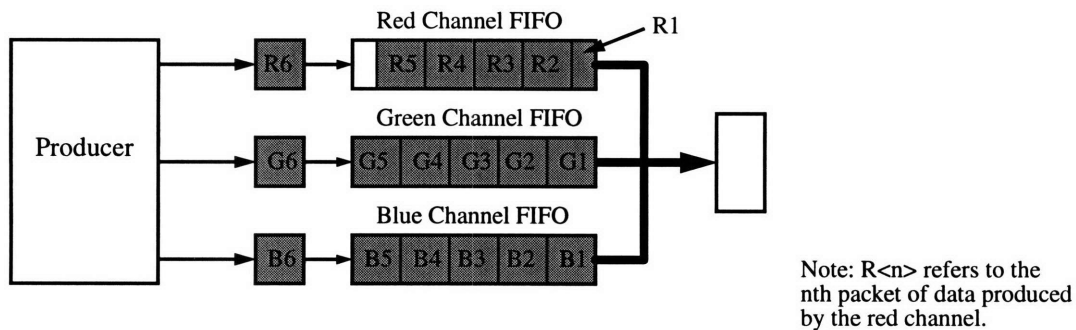


Figure 3-15: Packet Flushing Strategies for Producers with Symmetric Outputs.

In the above example, the auxiliary FIFOs on all three channels are becoming full at the same time that a packet is being emptied from the red channel FIFO. In order to avoid synchronization errors, the overflow manager FSM must clear the auxiliary FIFOs of the data packets R6, G6, and B6. Otherwise, if only G6 and B6 are cleared, R6 will be misaligned with G7 and B7, and the misalignment will continue for the duration of the stream.

3.3.6.3 Interfacing the Producer and FIFO - Multiple Asymmetric Outputs

If a producer generates several asymmetric outputs of varying throughput (such as three channels of video and two channels of audio), the aforementioned strategies of dealing with multiple streams will not work. If one uses an FSM which simply cycles through the channel FIFOs and waits for the selected FIFO to become full enough before initiating a transfer, the consumer will empty out the low throughput channel FIFOs before the high throughput FIFOs. This will eventually lead to a situation where the FSM will be stuck at a low throughput FIFO, waiting for sufficient data to initiate a bus transfer, while the higher throughput FIFOs overflow.

Instead, a demand-based consumer FSM is implemented which empties packets from several channels of FIFOs based on the relative fullness of each FIFO. If all of the FIFOs are equally full, higher priority is given to the lower throughput channels' FIFOs. Although the higher bandwidth channels' FIFOs are more likely to overflow in this case, this disadvantage is offset by the cost of losing an entire packet of from one of the low throughput channels. If such a loss were to occur, a potentially large portion of the producer's output could be affected, even though only one packet was lost.

A proposed alternate solution to this problem is to create heterogeneous packets which contain data from all the streams proportional to their relative throughputs; with this approach, the loss of a single packet would cleanly decimate all the channels for a shorter period of time.

Although this idea minimizes losses during overflow conditions, it remains a non-optimal solution. First, the storage of heterogeneous packets in the destination memory store would greatly increase the overhead involved in extracting data for separate channels at the next stage in the data processing pipeline. Second, rigidity in the packet format would prevent dynamic adjustments of individual producer outputs via flow control mechanisms, since such adjustments would change the portions of packets devoted to each channel of the producer's output. Without any formatting information within each packet, detection of such a transition would be impossible. Finally, once one considers the relative proportions of producer outputs in an actual slave device, one realizes that segmenting strategies become prohibitively complex in a majority of cases, with wasted bandwidth or non-uniform packet structures resulting from attempts to find a working heterogeneous packet structure.¹

3.4 Bus Master Overview

The bus master consists of several components: the bus interface, bus master arbiter, synchronizing register, header storage registers, FIFO block, and address generator/control logic block/serial interface.

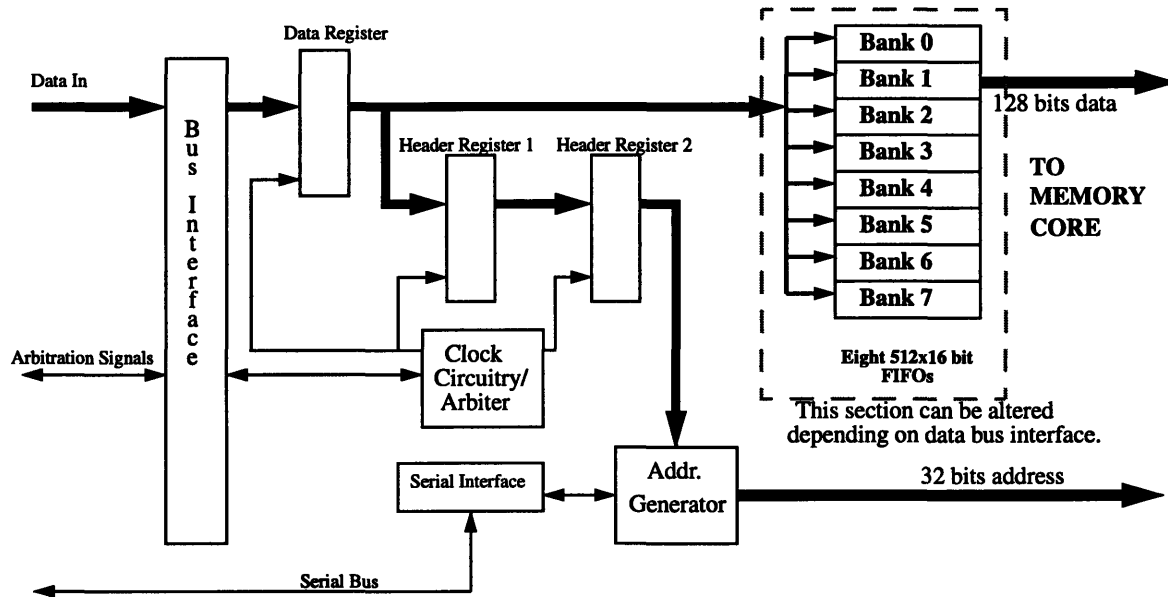


Figure 3-16: Bus Master Block Diagram

3.4.1 Bus Interface

The bus interface is identical to the upstream bus interface of a slave unit. It receives data, HV_CLK, and REQ from the closest slave unit, and it returns an ACK signal. All the necessary level translation and signal conditioning described in section 3.3.4 is performed in this unit as well.

3.4.2 Data and Header Registers

The bus master has one 16 bit data synchronization register placed after the bus

-
1. As an exercise, the author asks the reader to design a heterogeneous packet structure for a slave unit with (3) 640x480x30 video outputs and (2) 44.1 KHz audio outputs.

interface unit whose output can be directed to either the FIFO block or the first of two 16 bit header registers.

The header registers store the headers from incoming packets until the address generator has finished processing the currently serviced packet. In theory, the bus master could function without these registers by refusing to service another packet (not raising the ACK signal) until it has completed address generation on the present packet. However, such a design would cause the address generator to sit idle for a significant amount of time between each address generation while data propagated down the bus. By allowing a slave to send a packet to the bus master while the address generator is busy, the idle period between the processing of packets can, in a best case scenario, be reduced from approximately 7.5 μ s to tens of nanoseconds.¹

3.4.3 FIFO Block

The FIFO block is four packets deep to support the bus master's capability of accepting more than one packet. A three packet deep structure would have been sufficient, but such FIFOs are not available on the commercial market.

The actual width of the FIFO block is determined by the size of the memory store it interfaces to. If the FIFO block is wider than 16 bits (as most present day memory systems are), the incoming data is written in an interleaved fashion.

3.4.4 Arbiter

Unlike the slave, the bus master **arbiter** needs more status signals and internal

1. Assuming that 1) a non-pipelined bus master would have 1094 ns for roundtrip time from assertion of ACK by the bus master to the receipt of the first data byte plus an additional 6.4 μ s to transfer the remaining 512 bytes in the packet and 2) a pipelined bus master could receive the entire incoming packet before the address generator finished processing the previous packet.

variables to handle the queued up packets. There are three state bits and 14 input lines which are shown in the diagrams below, and one additional state machine described in the next section:

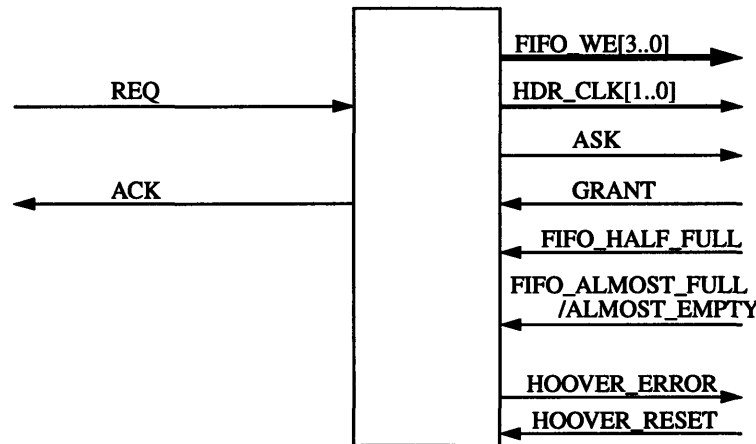


Figure 3-17: Bus Master Arbiter

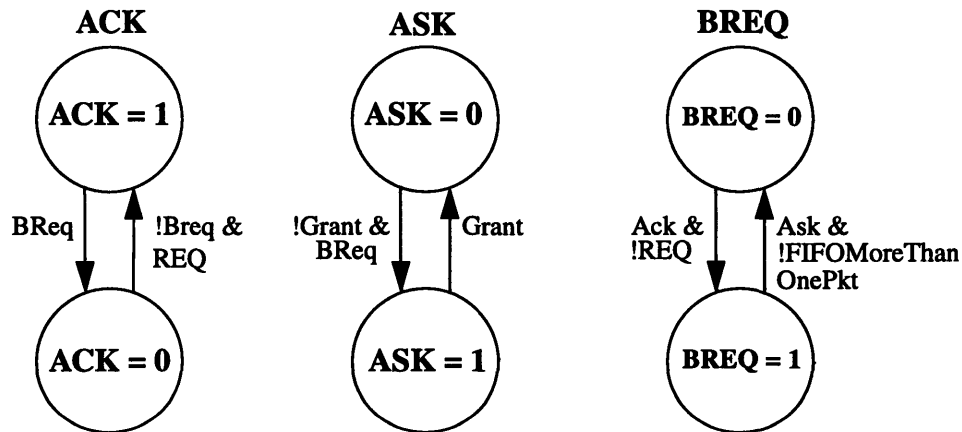


Figure 3-18: Bus Master Arbiter State Diagrams

The BReq state machine reflects the state of the header registers and FIFOs: the deasserted state means that *Header Register 1* currently does not contain a packet header; the asserted state indicates the presence of a packet header in the first header register. The BReq state machine transitions to the asserted state when the ACK state machine allows a packet to enter (by asserting the ACK line), and the actual data transfer is completed

(when the REQ line deasserts, indicating that the slave unit is done with the bus slot).

If this is the only packet in the bus master FIFOs (indicated by the !FIFOMoreThanOnePkt signal, which is a combination of the FIFO status flags), the packet header is transferred from *Header Register 1* to *Header Register 2* as soon as ASK is asserted.

However, in the case where there is more than one packet in the bus master, the ASK signal is still being asserted because the last packet has not been transferred to the memory core. Thus, the present header cannot be transferred until the previous packet is serviced, pushing the previous header out of *Header Register 2*. This action sets FIFO-MoreThanOnePkt false and deasserts the ASK signal. Once the ASK signal is reasserted, the header is transferred between registers, and BReq returns to the deasserted state.

The ACK state machine, as its name implies, controls the ACK output of the arbiter. It enters the asserted state when there is no header presently in *Header Register 1* and when REQ is asserted. It exits the asserted state when a header is placed into *Header Register 1* (see the description of BReq below).

The ASK state machine asserts the request line to the memory core arbiter when there is a packet header in *Header Register 1* and a transfer is not actively occurring. When the ASK state machine enters the asserted state, the clock line to *Header Register 2* is toggled, transferring the header from *Header Register 1* to *Header Register 2*. The FSM then waits for the memory core arbiter returns a single clock cycle strobe on the GRANT line, indicating an accessible core, at which point ASK moves to the deasserted state.

A timing diagram demonstrating the proper operation of the bus master arbiter is

shown below:

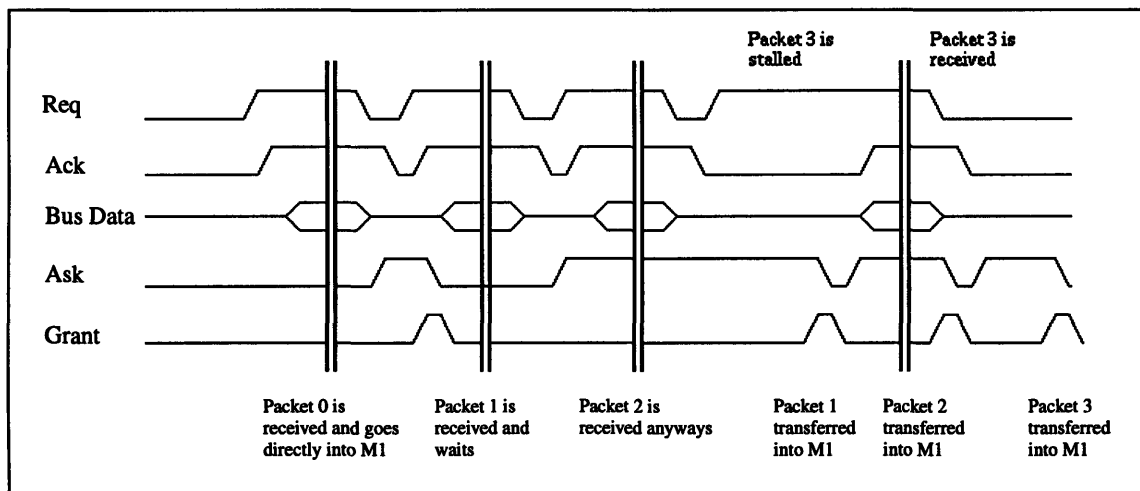


Figure 3-19: Hoover Bus Master Arbiter Timing
(adapted from an internal document “Hoover Bus Flow Control” by John Watlington)

3.4.4.1 Error Checking State Machine

The error checking state machine runs a counter tallying the number of words received by the bus master, and verifies that 256 words of payload have indeed been received for each packet. If the correct number of words has not been detected, the signal `HOOVER_ERROR` is asserted, and the address generator can clear the FIFO bank of the invalid data.

3.4.5 Clock Circuitry

Identical to the upstream clock convertor in the slave unit, the clock circuitry in the bus master converts the 40 MHz NRZ data clock into a 40 MHz return-to-zero clock signal used to clock the resync circuitry.

3.4.6 Address Generator

The address generator is a functional block, typically a microprocessor and mem-

ory subsystem or a hardwired control logic block with memory, which examines the header information of a packet from the synchronizing registers, and generates the proper memory core address to store the packet.

In order to select a device as an address generator, a bus designer must consider the rate of packet arrivals. For example, the calculations in Appendix A.2 indicate that in a worst case scenario, an address generator for a single Hoover bus can expect a new packet to arrive every 6.800 μ s. Thus, for a well written address generator kernel with an inner loop of 60 RISC instructions, one can expect relative success with even a non-superscalar RISC chip running at 20 MHz.

However, a input module with four fully saturated Hoover Buses can expect, on average, a new packet to arrive every 1.7 μ s. In such a case, a three-way superscalar RISC processor running at 20 MHz could theoretically be used for the address generator, depending on the structure of the kernel code. The kernel code should be tightly written to severely limit the slower, off-chip memory accesses, and data dependencies in the kernel code should be eliminated. In addition, all performance-enhancing features of an architecture (such as branch hints and data-caching) should be enabled.

Implementation

Chapter 4

A Hoover Bus consisting of two slave units and a bus master were designed from the design specifications outlined in the previous chapter. Rather than devoting a large amount of time and money to construct an entire system based on an initially unproven design, the test configuration was constructed as a separate, low cost, proof-of-concept addition to a larger, proven system with needs for Hoover bus functionality.

The bus master was included as a subsection of the I1 Input Card – a module designed and implemented for the Cheops Imaging System¹ by John Watlington, Hanoz Gandhi and the author. The I1 Input Card contains two Hoover bus interfaces and one video digitizer which converts either one NTSC or one RS-170A RGB 30fps signal into a deinterlaced, digitized 640x480x60 fps output stream. These sources interface to the M1 memory card - a 8 GB DRAM store - through a 400 MB/s local bus interface, which then makes the data accessible to the rest of the system.

1. Please see Appendix C for more details on the Cheops Imaging System.

4.1 Slave Implementation

The test slave units were required to be easily constructible and flexible enough to emulate a variety of different video sources. Thus, high performance, professionally fabricated digitizing slave units were not feasible. Instead, a purely digital test slave which stored input patterns in a small pattern buffer was proposed for system verification. Without an analog digitization section, the slave unit (see Figure 4-1 below) could be constructed with wirewrap techniques and be used to create a number of test bus configurations.

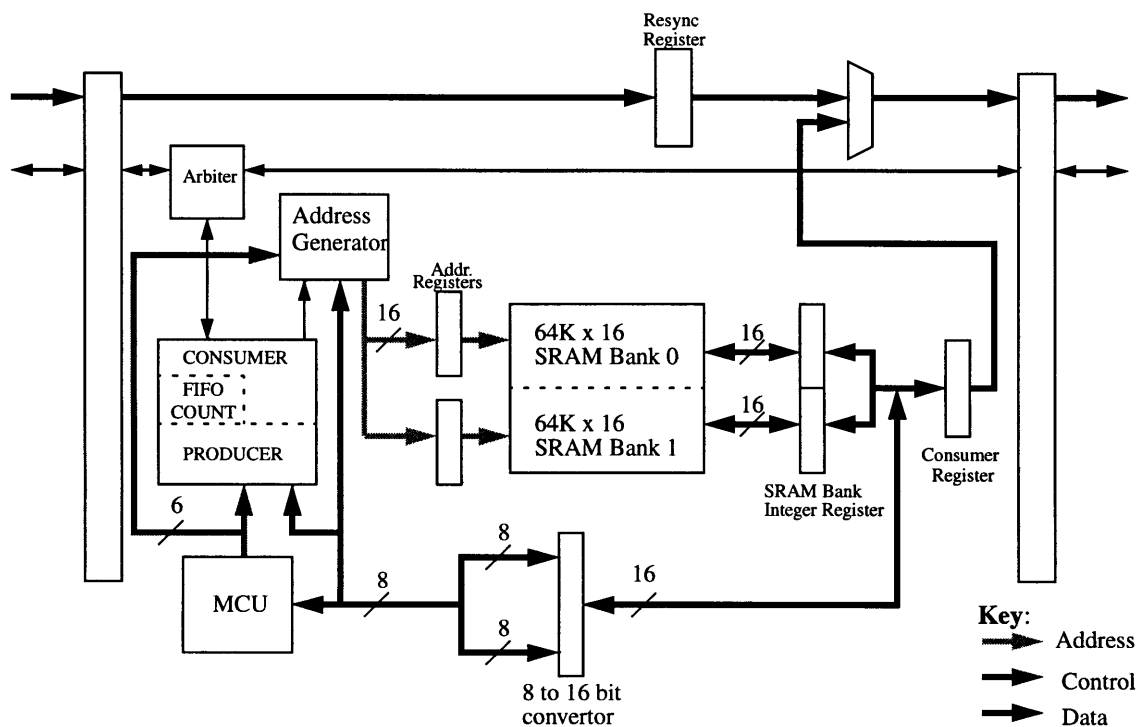


Figure 4-1: Test Slave Block Diagram

Two issues arose from the choice of simulating the producer as a static memory core containing the payload and header data. First, the SRAM and control circuitry necessary for true multichannel slave simulation was too complex for an inexpensive, “home-grown” test slave unit, as one would have to replicate much of the circuitry described in

the following sections for each channel. Thus, the decision was made to limit this test slave to single channel, multiple data rate outputs.

Second, the use of a static, predefined SRAM buffer enabled us to separate the FIFO data from the FIFO state. Thus, the producer needs only to alter the FIFO state variables to indicate the production of valid data; the consumer simply needs to detect the changed state variables and move the SRAM address pointers in response to this change. As a result, the test slave description will be broken into functional blocks which overlap the boundaries delineated in Chapter 3.

4.1.1 Slave Behavioral Description

The test slave is controlled primarily through the serial command set described in section 3.2. When powered up, the SRAM buffer is empty, and the various FSMs are in their idle state. The user can either configure the slave unit with one of the predefined test pattern/slave register sets, or by the modifying the individual registers to the desired values after a default set is chosen (one wants to use a predefined slave profile first to initialize the SRAM buffer; otherwise, one will have to manually program each word of the SRAM). After the slave is configured, one can then control hoover transfers from this slave unit by using the START_HOOVER and STOP_HOOVER commands in the serial command protocol.

4.1.2 MCU and Serial Interface

The MCU and serial interface provide a high level interface between the slave and the user. The serial interface consists of a Maxim MAX 491CPD RS-485/TTL transceiver connected to the built-in serial interface on the Motorola MC68HC11E2 MCU chip on the

slave unit. The receiver circuitry monitors the serial bus for messages with matching address ID fields and deposits the characters of a valid message to a slave core controller program running in the MCU.

4.1.2.1 Slave Core Controller Program

The core controller program is an event-driven program which cycles in a loop, waiting for input from the serial interface. Multicharacter messages are filtered through character-matching conditional statements until a valid message is extracted, or until a parse error is detected. If a valid message is detected, the program dispatches to the correct subroutine, and when done, returns to the parse loop. Otherwise, if a parse error is detected, a NACK message is sent back. Once inside the correct subroutine, the MCU extracts any needed operands from the remainder of the message and executes a sequence of consisting of one or more of the following:

- issues a command to the control logic EPLDs via a write to a command register.
- modifies a control parameter by performing a write to an EPLD register.
- writes a value to the SRAM buffer via a set of write registers.
- reads a value from the SRAM buffer via a set of read registers.

A flowchart of the core controller program and a memory map showing the locations of all the command and data registers are included on the next two page.

CORE CONTROLLER PROGRAM OUTLINE

BEGIN

Initialize stack and registers.
Initialize serial port.
Go into address wakeup mode.
Set slave ID from DIP switch.

PARSE_LOOP:

Wait for first address character present from serial port.
Go to ERROR_HANDLER if any errors detected.

Check lower bits to see if address character matches our slave ID.
If it does not, put serial port into address wakeup mode and go
back to address wakeup mode.

If ID matches, wait for next character on serial port.
Go to ERROR_HANDLER if any errors detected.

Does char match the first char of the first known message operand?
If so, wait for next character on serial port.
Go to ERROR_HANDLER if any errors detected.

See if second character matches second character of first known
message operand.

If it matches a known command, dispatch to correct subroutine.
If char does not match any known operand with identical first
character, go to PARSE_ERROR_HANDLER.

Does char match the first char of next message operand in known set?
{ go through same process again }

Does char match the first char of next message operand in known set?
{ go through same process again }

●
●
●

If does not match any known opcodes, go to PARSE_ERROR_HANDLER

END PARSE_LOOP

ERROR_HANDLER/PARSE_ERROR_HANDLER:

Send appropriate error message to Bus Master.
Return to Address Wakeup Mode.
Go to PARSE_LOOP.

END ERROR_HANDLER/PARSE_ERROR_HANDLER

Figure 4-2: Outline of Core Controller Program

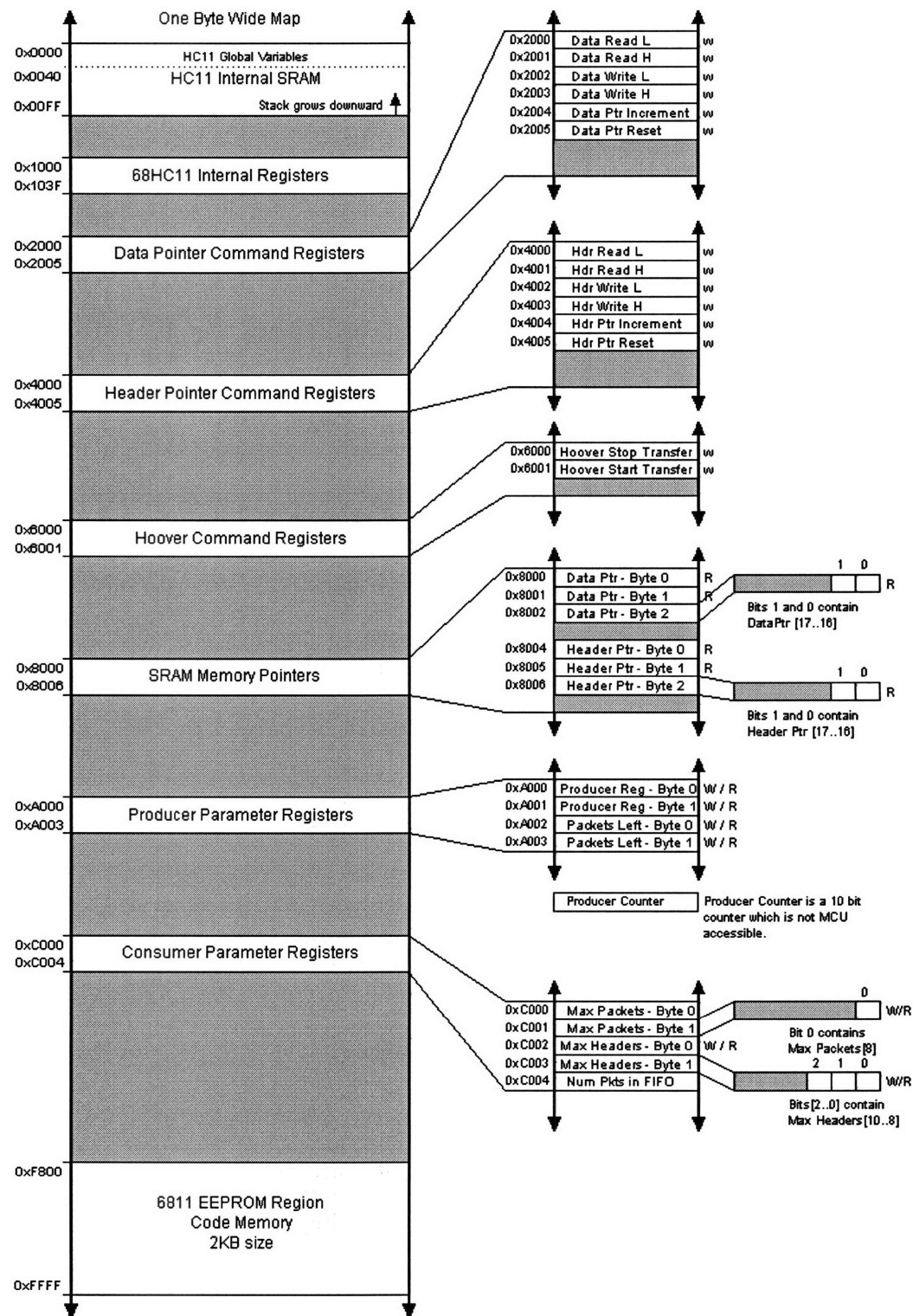


Figure 4-3: Register Map of Test Slave Unit

4.1.3 Producer/FIFO Buffer Status Indicators

The test slave producer and FIFO buffer status indicators consists of two separate logic blocks contained in a section of one of the Erasable Programmable Logic Devices (EPLDs) on the slave device: a producer FSM and FIFO position counter.

The producer FSM is a simple counter which asserts an incrementing signal, *pkts_in_fifo_inc*, when a number of clock cycles have transpired. The clock source is a 1.2288 MHz signal derived from the MCU clock, and the number of clock cycles is stored in a MCU writable register, *producer_reg*, on the EPLD. Simulation of different data sources is implemented by changing the value of *producer_reg*. For example, to simulate a single channel monochrome video digitizer output running at 640x240 x 60 fields per second, one would go through the following steps:

$$\begin{aligned} \text{bytes per field} &= 640 \times 240 &= &153600 \\ \text{bytes per second} &= 153600 \times 60 &= &9216000 \\ \text{packets per second} &= 9216000 / 512 &= &18000 \\ \\ \text{packet interarrival rate} &= 1 / 18000 &= &55.56 \text{ us} \\ \text{clock cycles per packet} &= 55.56 \text{ us} \times 1.2288 \text{ MHz} &= &68.27 \end{aligned}$$

Therefore, we can safely approximate the desired video source by writing 68 into *producer_reg*¹.

An additional control register which controls the duration of the operation of the producer FSM is *packets_left*, a 16 bit register which specifies the number of pkts to generate in this Hoover test run. Writing the maximum value, 0xFFFF, into this register is interpreted as an infinite duration bus simulation, and this register is not decremented.

-
1. Actually, video gets produced at a slightly faster rate during the fraction of time which corresponds to active video and is not produced at all during the remainder of stream time (blanking periods). However, because of the limited resources available on the EPLDs, this less accurate, but computationally simpler approach was implemented.

The *pkts_in_fifo* counter, a four bit FIFO status counter, is capable of simulating a 0 to 15 packet deep FIFO buffer through a 4 bit limit register, *max_pkts_in_fifo*. Upon the assertion of counter signal, the FIFO status counter increments by one to simulate the creation of one packet of producer information. If, due to backlog conditions, the producer tries to increment the FIFO position counter beyond the programmed allowable depth, an overflow signal is asserted and the counter stays at the full value.

4.1.4 Consumer and FIFO Data Circuitry

The test slave consumer is another finite state machine on the same EPLD as the producer and FIFO status indicator. It constantly polls the *pkts_in_fifo* counter, and upon detection of a non-zero value, asserts the WANT signal to the arbiter to begin the bus arbitration process.

Upon obtaining the SAFE handshake from the arbiter unit, the consumer begins extracting a header word and 256 payload words from the SRAM through two mechanisms: an external SRAM address generator which generates address bits [17..2] and a set of internal states which drive the lower address bits and associated control circuitry.

4.1.4.1 SRAM Pattern Buffer

In order to satisfy the slave performance requirements of 40 megatransfers/second onto the bus, the SRAM pattern buffer was constructed from two banks of 25 ns, 64K x 16 SRAM devices. Although an interleaved memory store increased the amount and complexity of control logic circuitry and datapaths in the slave device, the expense and availability of sufficiently fast (12 ns) SRAM made a non-interlaced design impossible to accomplish in the time constraints of this project.

As Figure 4-4 on the next page shows, the memory is partitioned into two separate regions - a 252 kilobyte (504 packets) data region occupying addresses 0x0000 to 0x3EFFF and a 4096 byte header region occupying addresses 0x3F000 to 0x3FFFF. This separation of SRAM regions, in conjunction with independent addressing mechanisms for each region, yields a larger number of possible simulated slave streams.

The ordering scheme for the packet data words and headers words is designed to ensure that packets can be pushed onto the bus at 40 MHz without any timing problems. Packets are divided into two classes: even packets and odd packets. Data for even packets (packet 0, packet 2, packet 4, etc.) are arranged in a conventional byte ordering. Thus, packet 0 has word 0 located at byte addresses 0x0 to 0x1, word 1 located at byte addresses 0x2 to 0x3, word 2 located at addresses 0x4 to 0x5, and so forth. Odd packet data and the header region, however, have their even and odd words swapped. Thus, word 0 for packet 1 is placed at addresses 0x203 to 0x202, and word 1 is placed at addresses 0x201 and 0x200, etc.

The motivation for this design is simple. Since the same bank cannot be accessed two times in a row at 40 MHz, one must make sure that the header word and the first data word lie in different banks. Thus, if header word 0 lies in bank 0, the first word of its payload must be in bank 1. Similarly, if header word 1 lies in bank 1, the first word of its payload byte must be in bank 0. In order to have word 0 of packet 0 start at address 0x0 though, it was necessary to swap the even and odd words in the header region.

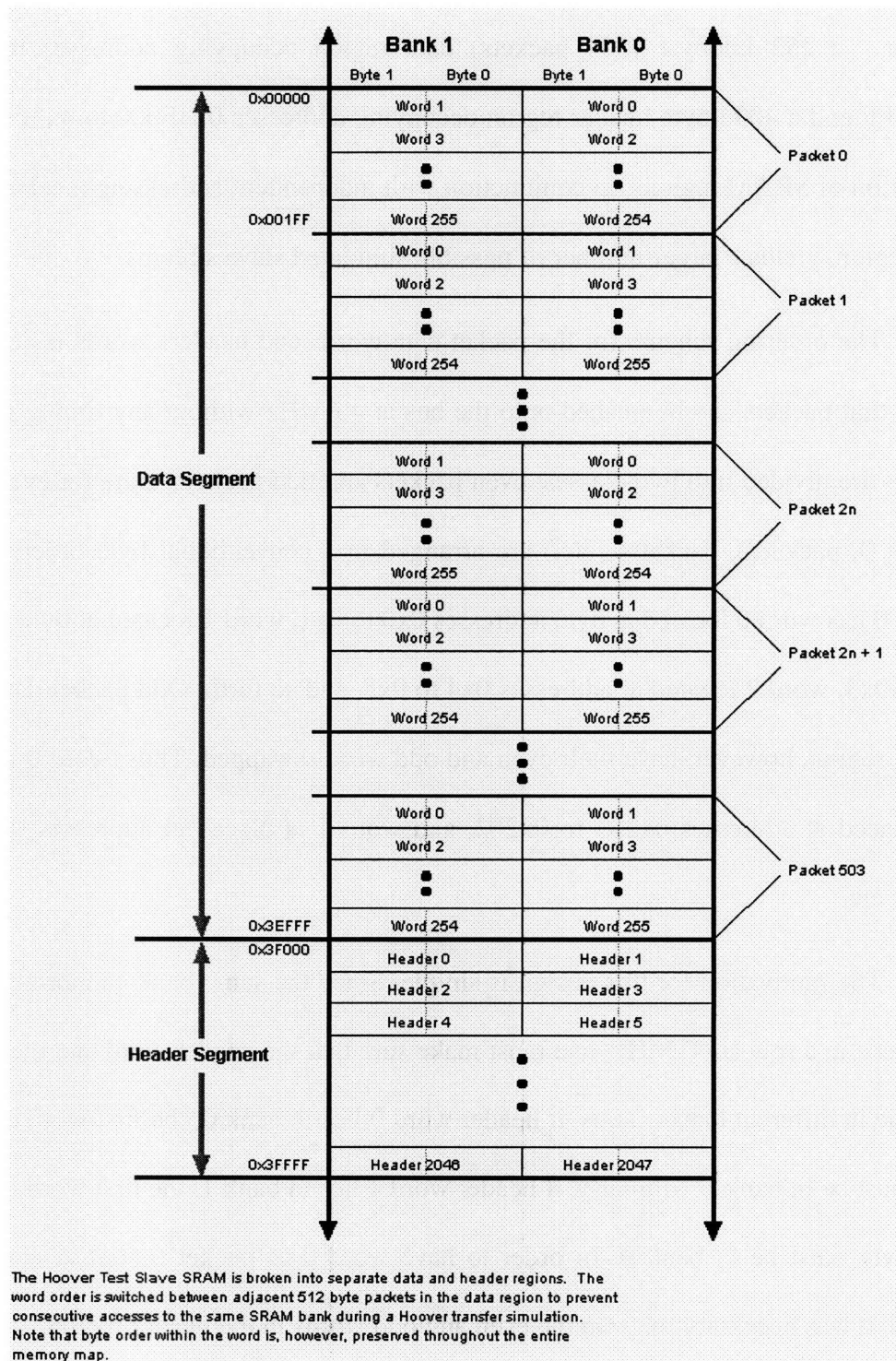


Figure 4-4: Hoover Test Slave SRAM Buffer Diagram

4.1.4.2 SRAM Address Generator

The SRAM address generator intelligently traverses the SRAM buffer when the

consumer deposits a packet onto the bus or when the MCU programs pattern values into the SRAM region. In order to accomplish this task, a number of counters and registers are required.

At the heart of the SRAM address generator are two address state machines: *data_ptr* and *hdr_ptr*, which keep track of the currently accessed *longs* (4 byte quantities) in their respective SRAM regions. The *hdr_ptr* is a ten bit incrementer which only computes address[11..2] since address[17..12] are always 0x3F for the header address. When incrementing, the counter advances the address pointer to the upper limiting address specified by the register *max_packets*, and then returns to the base address 0x3F000. The *data_ptr* is a larger 16 bit incrementer which, like the *hdr_ptr*, has its upper bound stored in an associated register, *max_packets*, and returns to its base address, 0x00000, when incrementing past this value.

Both state machines can be incremented by one of two methods. First, the consumer can assert increment commands at up to 40 MHz via two control lines: *data_ptr_inc* and *hdr_ptr_inc*. In addition, the MCU can increment the header and data pointers via slower MCU writes increment command registers.

4.1.4.3 Producer and MCU SRAM Control Circuitry

When the producer has successfully arbitrated for the a bus slot and begins to send a packet onto the bus, the producer uses the *hdr_ptr* and *data_ptr* to manage the upper 16 address bits of the SRAM pattern buffer. However, the producer must also control the lower two address bits, changing the order of sequential bank accesses depending on if the packet is even or odd.

As a result, a packet parity state machine which remembers the parity of the current packet is included in the producer to aid in the generation of the lower address line values. However, test slave hoover transfers must obey the following constraints to not confuse the packet parity FSM:

- hoover transfers must start on an even packet
- the upper bounds on the data and header regions **MUST** be even numbers

Note that the MCU does not have to obey the above rules when it accesses the SRAM pattern buffer. As a result, the parity state machine is unable to determine the correct ordering of the lower two address bytes when the MCU is writing to or reading from the SRAM pattern buffer.

To ensure proper lower address line management during MCU SRAM operations, the address line `addr[9]` is fed into the MCU's SRAM read/write circuitry. During operations on even packet regions, `addr[9]` will be zero; conversely, when accessing the odd packet regions, `addr[9]` will be one. Thus, by monitoring the status of this line, the EPLD can successfully manage the lower address lines during MCU SRAM reads and writes (Note that this technique could not be used during the much faster bus transfers, as the propagation delay of the address line would have caused timing problems with the SRAM control signals).

4.1.5 Arbiter/Upstream Clock Convertor/Downstream Clock Convertor

The arbiter, upstream clock convertor, and downstream clock convertor were implemented by translating the specifications described in Chapter 3 into Altera's Hardware Description Language and programming the resulting image into another EPLD.

4.1.6 Bus Interface Unit

On the test slave units, the active termination circuitry for the data and control lines is implemented with several discrete components and two op-amps. A stable voltage reference is provided with a reverse-biased zener diode. A voltage divider then yields a termination voltage value from the reference circuit. The resulting value is fed into two op-amp follower circuits with inline current amplifiers. Each of these active termination voltage sources is then heavily bypassed and connected to the 33 ohm termination resistors on the signal lines.

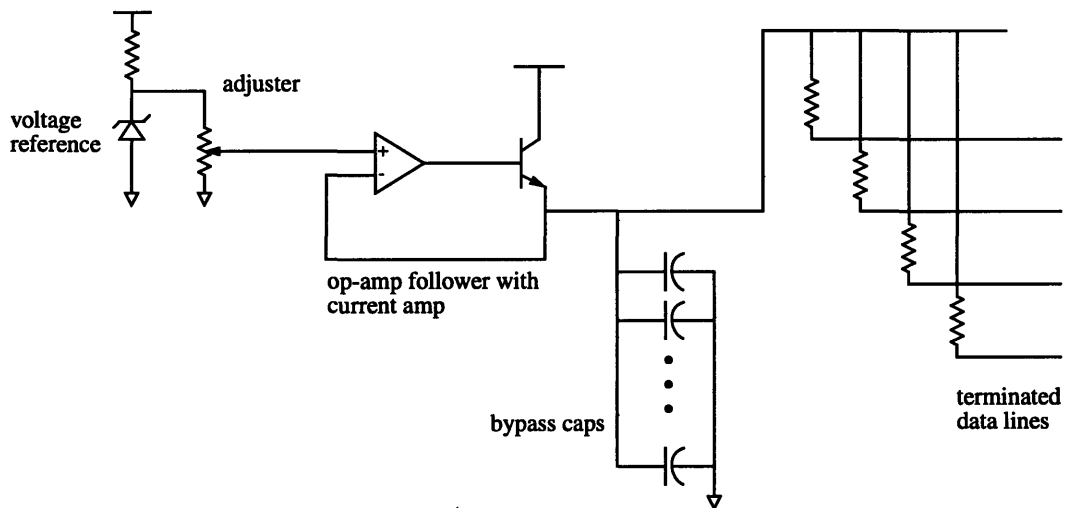


Figure 4-5: Slave Termination Circuitry

The terminated signals are then fed into BTL transceivers to switch to TTL levels. The data lines are sent through Texas Instruments 75ALS056 octal BTL transceivers, and the control/clock signals are sent through a 75ALS057 pin-programmable BTL transceiver.

4.2 Bus Master Implementation

4.2.1 Memory Store Unit

The memory store unit which interfaces to the bus master FIFOs is the M1 Memory Card - a custom board designed for the Cheops Imaging System which is capable of holding up to 8 GB of standard 72 SIMMs. Data transfers from daughtercards are sent across a 300 pin local bus connector containing 128 data lines, 29 address lines, several control signals and a separate Global bus interface for slower communications purposes. Transfers are currently limited to 512 byte bursts which occur at a rate of 1 quadword (16 bytes) per 31.25 ns.

The memory store interfaces to the rest of the Cheops Imaging System through two data buses and one control bus. Two high speed Nile buses capable of 120 MB/s sustained throughput handle large data transfers, and the aforementioned Global bus provides a register interface to the general purpose configuration microprocessor in the P2 processor module.

4.2.2 Address Generator

An Intel i960 superscalar RISC processor running at 32 MHz with an external 4 MB, 20 ns SRAM module comprises the core of the address generator. The address generator is surrounded by support circuitry implemented with Altera EPLDs which interface

the address generator to the remainder of the I1 Input Card.

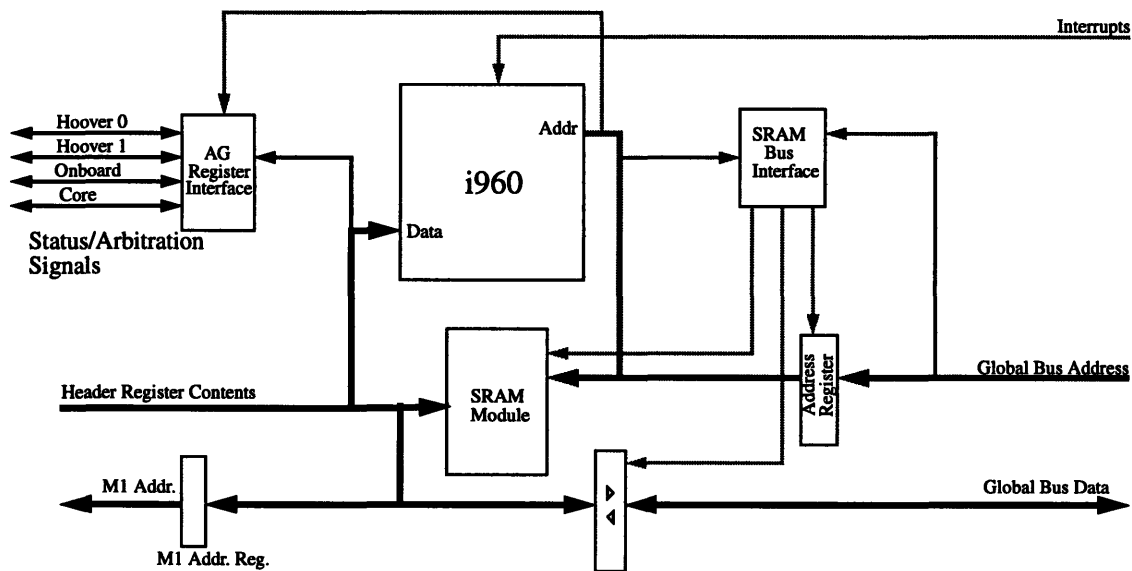


Figure 4-6: Address Generator and Support Circuitry

Status signals from each of the data producing subsystems on the I1 Card (Hoover Bus 0, Hoover Bus 1, and the onboard digitizer) are used to generate register values in the *AG Register Interface* EPLD. The address generator polls these registers to determine which data producing mechanisms on the I1 Card have enough data to transfer to the M1 Memory Card. If a subsystem has sufficient data, the address generator arbitrates for an M1 memory core access through the *AG Register Interface* and computes an appropriate address based on either incoming header register contents (for Hoover Bus packets) or internal variables (for the onboard digitizer packets).

The kernel code is initially loaded into the SRAM module by the external host system via the Global Bus prior to the activation of the address generator. As the kernel code is run the first time, the code gets transferred into the internal i960 instruction cache for faster performance. The *SRAM Bus Interface* EPLD ensures that both the Global Bus and address generator have a correctly timed interface to the SRAM and that simultaneous

attempts to access the SRAM are arbitrated between the devices in a non-destructive manner.

4.2.3 Bus Interface Unit

The bus interface unit for the bus master is implemented through drop-in components which matched the performance requirements of the Hoover bus design. Active termination is accomplished through two DS-2107A active SCSI terminators manufactured by Dallas Semiconductor. Electrical signal translation, as on the slave unit, is accomplished with TTL/BTL transceivers from Texas Instruments – 75ALS056 octal transceivers for the data lines, and a 75ALS057 bidirectional transceiver for the control and clock lines.

4.2.4 Bus Master Arbiter and Clock Circuitry

The bus master arbiter and clock conversion circuitry are implemented as firmware in Altera EPLDs. The arbiter and clock conversion circuitry are exact translations of the designs presented in the bus master architecture section, and are thus not described in any greater detail in this section.

4.2.5 FIFO Module

The bus master design specifies the FIFO block for the I1 Input Card's Hoover bus master to be four packets deep. Since the memory core interface for this system is 128 bits wide, a 64 quadword (64x128) FIFO was used. Texas Instruments' 74ACT7813 asynchronous 64x18 FIFOs were chosen as a building block for this section because of their relative low cost, availability, and performance specifications. They met the minimum operating frequency of 40 MHz and had programmable status flags which could be used to

signal the level of fullness/emptiness of a FIFO.

4.2.6 Serial Interface

Serial communication hardware was implemented through a Motorola MC2681 DUART (dual universal asynchronous receiver/transmitter) and two MAX491CPD BTL/TTL transceivers. The 2681 DUART was chosen for this platform because of its ability to handle two separate serial buses, act as an interrupt manager for up to six incoming interrupts, and drive up to seven independent outputs.

One consequence which arose from the choice of this component on the I1 Input Card was the discovery that baud rates on the DUART were limited to 38.4 Kbaud¹, thus limiting the speed of the serial command protocol.

4.3 Software Interfaces

Several programs and software modules were created to support the Hoover bus hardware. They include slave utility programs, slave parser programs, and serial interface libraries.

4.3.1 Serial Port Interface Software

A serial port interface was written to establish communication between the Cheops Imaging System application layer and the serial interface hardware on the Hoover bus master. The primary function calls used by the user are

```
void il_send_to_uart_nb(  InputSource which_source,
                        int which_uart,
                        unsigned char *message_in,
                        int message_in_length);
```

1. This limitation can be overcome by deriving a faster clock from an auxiliary frequency source. However, the I1 Card had been fabricated at this point, and thus could not be modified to utilize the auxiliary frequency source.

```
void il_send_to_uart_with_reply_nb(  InputSource which_source,
                                     int which_uart,
                                     unsigned char *message_in,
                                     int message_in_length,
                                     unsigned char *message_out,
                                     int message_out_length);
```

and

```
int il_send_to_uart_with_indexed_reply_nb(InputSource which_source,
                                           int which_uart,
                                           unsigned char *message_in,
                                           int message_in_length,
                                           unsigned char *message_out);
```

These function calls, as the “nb” in their name suggest, are non-blocking function calls which send messages to the slave units from the Input Card identified by the variable `which_source`.

The first variant, `il_send_to_uart_nb()`, sends a message located in `message_in` of length `message_in_length` onto the bus without looking for a reply from the slave unit. This command is used largely for diagnostic purposes and was constructed initially as a test code segment to base the other two function calls upon. The second function, `il_send_to_uart_with_reply_nb()` sends a message in the same manner as the first function, but looks for a reply of length `message_out_length` on the UART receiver, and places the result in the buffer `message_out`. This function call is used where a reply of known length will be returned from the addressed slave. One such place where this function call is used is in the slave uploader utility *program_6811* (see section 4.3.2. for more information on this program), where for each serial character sent to the slave unit in bootstrap mode, an identical serial character is sent back to the transmitting unit as a handshake signal.

The third function, `il_send_to_uart_with_indexed_reply_nb()`, sends out a message in the same manner as the two previous function calls, but instead expects back a

Hoover slave reply message whose length is encoded into the packet as described in section 3.2.6.

Each of these aforementioned functions invoke a large number of lower level library calls which manage semaphores, interrupts, and a large number of bookkeeping variables. However, an in-depth discussion of the execution of library calls in these functions is beyond the scope of this thesis.

4.3.2 Uploading Slave Core Controller Code

The Hoover bus slave's core controller program is written in 6811 assembly code and needs to be burned into the slave's 68HC11 MCU EEPROM. To accomplish this, the source file is first assembled with the Motorola freeware assembler *as11**. The output of this program is an S19 file which contains an ASCII version of the 6811 machine code. Although S19 files are supported by IBM PC uploaders, the Cheops serial libraries do not support such an option. Thus, in order to upload the machine code program onto the Hoover slave device, several steps are required. First, the utility *convertor*^{*1} is used to generate a binary image of the S19 code. Next, the existing slave chain is powered-down and disconnected from the bus master; the unprogrammed slave unit is then connected to the bus master as the sole device on the bus. The slave unit's mode switch is then set to bootstrap mode, and both the Cheops unit and slave device are powered on.

At this point, *program_6811* is now run on the Cheops Imaging System to upload the binary image file into the 6811 MCU EEPROM. This program uses the 68HC11 bootstrap mode – a special mode which receives, stores into internal RAM, and executes a 256

1. Full source code for *convertor.c* and other non-Cheops specific 68HC11 utilities are included in Appendix D.

byte assembly program from the serial line – to upload an auxiliary EEPROM programmer routine. This EEPROM programmer routine is a modified version of the internal bootstrap routine which stores a variable length assembly program into the 68HC11 EEPROM instead of into internal RAM. After the EEPROM programmer routine begins executing, the *program_6811* utility begins uploading the user's slave core controller code into the MCU EEPROM. Once the entire core controller is in place, the EEPROM programmer utility resets the program counter to point to the base of the EEPROM region and begins executing the core controller code.

Conclusions

Chapter 5

After the necessary subsystems of the test Hoover Bus configuration were constructed, the debugging process began. However, design flaws with the onboard digitizer circuitry and I1 kernel code in conjunction with other unforeseen logistical difficulties left insufficient time for the proper testing and verification of the test Hoover Bus. As a result, the decision was made instead to produce a complete set of specifications documenting the present version of the bus protocol and to provide an analysis of the relative strengths and weaknesses of the existing system, so that further development and improvement can be made by future researchers.

5.1 Current State of the Test Bus Configuration

Currently, all the necessary components of the test bus configuration – test slaves, bus master, and software library – have been constructed. Extensive simulation of the high speed core circuitry (producer, FIFO status indicators, consumer, and SRAM address gen-

erators) on test slave device with the Altera Programmable Logic Development System has left the author confident of the correctness of this section of the test slave design. In addition, the process of debugging the onboard digitizer/deinterlacer circuitry has uncovered a number of design flaws on the I1 Input Card which would have lain in the critical path of obtaining Hoover Bus functionality.

However, at this time, the serial command subsystem of the Hoover Bus is not functioning reliably, which prevents the proper uploading of the slave core controller code into the slave unit and prevents functional testing of the system. The design and implementation of the 68HC11 support utilities, two stage uploader program, and serial communication libraries for a multitasking OS proved to be more difficult than initially expected. Nevertheless, the majority of the bugs in this system have been fixed, so the author feels that functionality of this subsystem could be achieved with a modest amount of work.

Once the serial subsystem is operating, existing diagnostic applications written for the I1 onboard digitizer can be modified to fit the Hoover Bus configuration, and the necessary experiments can be run to determine the soundness of the Hoover Bus design.

5.2 Evaluation of Current Design and Recommendations for Future Work

After re-evaluating the entire Hoover Bus design and the engineering tradeoffs made during the design process, the author has come to several conclusions.

First, implementing the Hoover Bus with a serial topology instead of a parallel topology was a good design choice. The skew and synchronization problems associated with a parallel topology would have made the implementation of a signalling layer signifi-

cantly more expensive and difficult. However, the failure of a single node in a serial topology bus will render the entire upstream segment useless; further research should be done to examine the feasibility of using redundant topologies which can tolerate the loss of a single node without the loss of functionality for a large section of the bus.

Second, the choice of using a 16 bit, 40 MHz data transmission channel proved to be optimal for the present implementation of the Hoover bus. Datapath widths greater than 16 bits would have made routing of connectors more difficult and would have required a significant amount of additional hardware at each slave node. 40 MHz proved to be a good operating speed for the datapath and associated control logic. The complexity of the EPLDs in the test slave units reduced the operating speed of the core control logic to 41.66 MHz.

Nevertheless, the choice of a 16 bit, 40 MHz data transmission channel meant that a maximum throughput of 80 MB/s was not attainable once arbitration time was considered. In fact, the calculations in Table 3 show that over 10 MB/s can be lost due to arbitration overhead when using the 512 byte packet payloads as specified by the current specification of the Hoover Bus.

As a result, the author suggests that further research needs to be done to find a way to increase usable bandwidth on the Hoover bus. The use of faster, simpler EPLD control logic circuits could theoretically boost the data clock to over 50 MHz, and the use of 1024 byte packets is no longer prohibited by the lack of high capacity, high-speed commercial FIFOs.

Third, further research needs to be done to evaluate the benefits and costs of add-

ing error detection for slave-to-slave data transfers. Presently, the only error detection is done at the bus master by the payload word counter described in section 3.4.4.1. If the number of words coming into the bus master does not equal 256, the packet is rejected.

The above approach, however, wastes valuable physical resources when a corrupted packet is sent needlessly to the bus master, when it may have been corrupted on the first point-to-point transmission. Future researchers should examine the error rates on point-to-point links and make a recommendation on whether or not such error detection is worth the additional control logic needed to implement it. Furthermore, if error detection is deemed necessary, work should be done to determine if error *correction* is necessary. The idea of a resend FIFO – a packet sized FIFO which stores the last transmitted packet until correct receipt has been acknowledged and retransmits if necessary – has been discussed, but the benefits of such a measure may outweigh the costs of implementing it.

Appendix A: Bus Transfer Timing Calculations

A.1 Average Handshaking Overhead

In Table 3 in section 3.1.3, the quantities *typical average handshaking overhead* and *worst case average handshaking overhead* are used. Both quantities refers to the average amount of time in a master-slave Hoover data transfer that is not used for actual data transfer. The terms *typical* and *worst case* refer to the topologies of the two hypothetical buses used to calculate the respective overhead amounts.

There are two parts of this overhead: the first, *average arbitration latency*, is the amount of time that it takes for a slave to send a request to the bus master and receive an acknowledge; the second, *average deassertion latency*, is the time it takes from the slave's initial indication that it is done with the bus (the deassertion of its ACK signal) to the time when the another slave unit can begin arbitrating for the bus.

For the purposes of this study, the average deassertion latency is assumed to be the period of time from the slave's initial deassertion of its REQ signal to point in the time when the bus master's deasserted ACK signal reaches the active slave; this assumption makes the average deassertion latency identical to the average arbitration latency.

In reality, the average deassertion latency can be shorter than our estimated value. If the deasserted REQ signal has passed through a slave, the slave will assume that it can arbitrate for the bus, even though the deasserted REQ signal may have not reached the active slave yet. However, it is impossible to accurately compute an average deassertion latency without more sophisticated models of the slaves' producers and FIFO configurations.

A.1.1 Average Overhead for Typical Bus Configurations

For the typical Hoover bus configuration, average handshaking overhead was calculated by doubling the average arbitration latency for the third slave in a Hoover bus with slave-to-slave connections made via 1.5 meter, synchronous transfer cables as defined in Section 4.2.3. of the ANSI SCSI-2 Standard.

Downstream Propagation Delay

active slave's consumer - arbiter delay	25 ns
active slave downstream driver delay	18 ns
cable delay (6.0 meters)	33 ns
slave node relay delay (2 nodes)	2 x 73 ns
(40 ns upstream receiver delay +	
10 ns clock convertor delay +	
5 ns register output delay +	
18 ns downstream driver delay)	
bus master upstream receiver delay	40 ns
bus master arbitration delay	50 ns

Total	312 ns

Upstream Propagation Delay

bus master upstream delay	18 ns
cable delay (6.0 meters)	33 ns
slave node relay delay (3 nodes)	2 x 73 ns
(40 ns upstream receiver delay +	
10 ns clock convertor delay +	
5 ns register output delay +	
18 ns downstream driver delay)	
active slave's downstream receiver	40 ns

Total	237 ns

The total arbitration delay is therefore the sum of the propagation delays: 547 ns, and the total handshaking overhead is twice this amount: 1094 ns.

A.1.2 Worst Case Average Arbitration Time

For the worst case Hoover bus configuration, handshaking overhead was calculated

in a similar fashion for the fourth slave in a Hoover bus with slave-to-slave connections made via 6.0 meter, synchronous transfer cables as defined in Section 4.2.3. of the ANSI SCSI-2 Standard.

Downstream Propagation Delay

active slave's consumer - arbiter delay	25 ns
active slave downstream driver delay	18 ns
cable delay (24.0 meters)	132 ns
slave node relay delay (2 nodes)	3 x 73 ns
(40 ns upstream receiver delay +	
10 ns clock convertor delay +	
5 ns register output delay +	
18 ns downstream driver delay)	
bus master upstream receiver delay	40 ns
bus master arbitration delay	50 ns
-----	-----
Total	484 ns

Upstream Propagation Delay

bus master upstream delay	18 ns
cable delay (6.0 meters)	132 ns
slave node relay delay (3 nodes)	3 x 73 ns
(40 ns upstream receiver delay +	
10 ns clock convertor delay +	
5 ns register output delay +	
18 ns downstream driver delay)	
active slave's downstream receiver	40 ns
-----	-----
Total	409 ns

The total arbitration delay is therefore the sum of the propagation delays: 893 ns, and the total handshaking overhead is 1786 ns.

A.2 Maximum Packet Interarrival Rate

To accurately choose an address generator, one needs to calculate the maximum packet interarrival rate. For a single bus, a safe estimate for the maximum packet interarrival rate can be derived by calculating the total transfer time for a single slave unit hooked

up to the bus master with a one meter cable. Thus, for a 512 byte packet size, the separate components of arbitration delay is

Downstream Propagation Delay

active slave's consumer - arbiter delay	25 ns
active slave downstream driver delay	18 ns
cable delay (1.0 meter)	5.4 ns
bus master upstream receiver delay	40 ns
bus master arbitration delay	50 ns
-----	-----
Total	138.4 ns

Upstream Propagation Delay

bus master upstream delay	18 ns
cable delay (1.0 meter)	5.4 ns
active slave's downstream receiver	40 ns
-----	-----
Total	63.4 ns

and the total arbitration delay is 202 ns.

The total handshaking overhead is therefore 404 ns; the total transfer time per packet is 6400 ns + 404 ns = 6804 ns; and the maximum packet interarrival rate is 146972.37 packets/second.

Appendix B: Termination Techniques

As discussed in section 3.1.1.5, the objective of termination is to minimize voltage reflections along a transmission line with characteristic impedance Z_o by providing an AC impedance of value Z_o to ground at the far end of the transmission line.

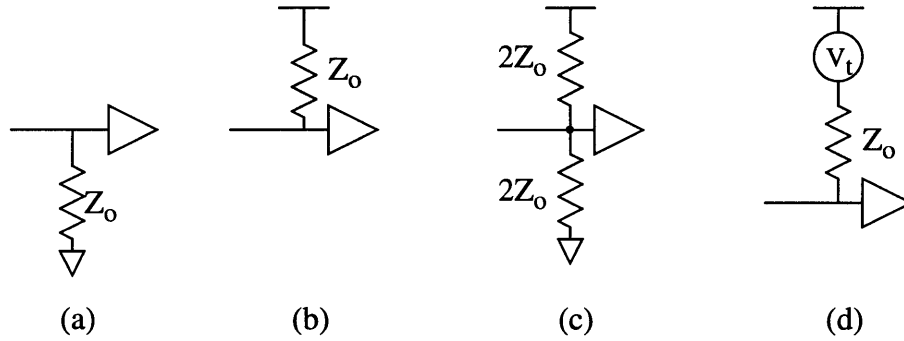


Figure B-1: Different Types of Termination.

Figure (a) shows an example of DC termination to GND. Figure (b) shows DC termination to Vcc. Figure (c) is an example of Thevenin termination. Figure (d) shows an implementation of active termination.

The simplest method of implementing this is called DC termination. One terminal of a termination resistor is placed at the far end of the transmission line; the other terminal is connected to ground (see Figure B-1(a)). However, as noted before in section 3.1.1.5, this approach consumes $\frac{V_I}{Z_o}$ amps of current when the driver is sending a logic one. Thus, it follows that for a signal with duty cycle k , the average power consumption P_T for this method is

$$P_T = k \frac{(V_I)^2}{Z_o} \quad (\text{B - 1})$$

As one can see, the power consumption is proportional to the duty cycle of the signal. Although this method works well for low duty cycle signals, it burns excessive power

for signals that are mostly at Vcc. For these signals, one can choose a DC termination to Vcc (shown in Figure B-1(b)) instead, whose average power consumption is

$$P_T = (1 - k) \frac{(V_I)^2}{Z_o} \quad (\text{B} - 2)$$

If the circuit designer knows the characteristics of the signal, he or she can choose the most appropriate termination method to minimize power consumption. Most of the time, however, the system designer does not want to spend the excess time characterizing the behavior of the signals in a system.

Thevenin termination avoids this problem by using two termination resistors: a resistor of resistance $2Z_o$ between the transmission line and Vcc, and another $2Z_o$ resistor between the transmission line and GND (see Figure B-1(c)). If an AC analysis of this circuit is done, one can see that the termination impedance to GND is still Z_o .

However, the power consumption calculations are slightly different. For an arbitrary signal $x(t)$ sent down the transmission line, we can state that it will spend some fraction k of its time in the high state, and spend an additional time $(1 - k)$ in the low state. For the portion of time $x(t)$ is high, the power dissipation P_{T_H} will be solely across the lower resistor. Conversely, when $x(t)$ is low, the power dissipation P_{T_L} will be across the upper resistor. By inspection, one can see that

$$P_{T_H} = P_{T_L} = \frac{(V_I)^2}{2Z_o} \quad (\text{B} - 3)$$

The total power dissipation P_T can then be derived by multiplying the separate power dissipation values by the respective amounts of time that the signal is each state. Thus,

$$P_T = k(P_{T_H}) + (1 - k)(P_{T_L}) = k \frac{(V_I)^2}{2Z_o} + (1 - k) \frac{(V_I)^2}{2Z_o} \quad (\text{B - 4})$$

which simplifies to

$$P_T = \frac{(V_I)^2}{2Z_o} \quad (\text{B - 5})$$

Thus, the power consumption of a Thevenin termination circuit is always the equivalent to that of a DC termination circuit with an input signal with a duty cycle of 50%, regardless of the actual duty cycle of the input signal. Thus, the circuit designer can use the Thevenin termination circuit and be assured that the power dissipation for the terminator will be constant, whereas the designer using DC termination will have to ensure that their power supply can withstand a worst case power drain that is twice as high as the expected average power draw for the circuit.

Active termination (Figure B-1(d)) further improves upon the Thevenin termination circuit by replacing the resistor bridge with a resistor of $Z_o \Omega$ and an independent voltage source set to some intermediate voltage between Vcc and GND, usually $\frac{V_{cc}}{2}$. If one repeats the derivation steps from the Thevenin equivalent example, one finds that

$$P_{T_H} = P_{T_L} = \frac{(V_I)^2}{4Z_o} \quad (\text{B - 6})$$

and the total power dissipation P_T becomes

$$P_T = k(P_{T_H}) + (1 - k)(P_{T_L}) = k \frac{(V_I)^2}{4Z_o} + (1 - k) \frac{(V_I)^2}{4Z_o} = \frac{(V_I)^2}{4Z_o}. \quad (\text{B - 7})$$

Thus, active termination uses half the power of the Thevenin equivalent termination circuit

at the cost of adding in an independent power supply.

Active termination is preferred for buses with many signal wires for two reasons. First, the lower power consumption savings become quite significant as the signal line count increases. Second, by using one independent power supply for all the signal lines, one can actually reduce the component count and space requirements for a bus terminator when compared to the Thevenin termination circuit, which requires twice the number of resistors.

Appendix C: The Cheops Imaging System

The Cheops Imaging System is a compact, inexpensive reconfigurable dataflow architecture designed for experiments in topics such as model-based video encoding, spatial imaging, and open-architecture video [1]. Instead of performing computations with one or more high-performance general-purpose processors in a traditional von Neumann architecture system, the Cheops system uses a dataflow architecture which routes data through *stream processors* - small, inexpensive modules which perform the computationally expensive video processing operations in dedicated hardware.

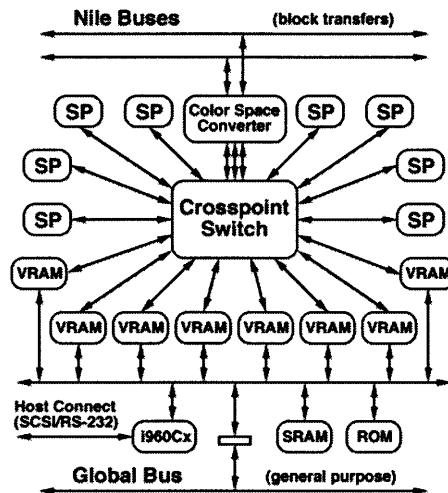


Figure C-1: Block Diagram of P2 Processor Card

(figure and caption from *Cheops: A Reconfigurable Data-Flow System for Video Processing*) Highly simplified diagram of the Cheops processor. Blocks labeled SP represent stream processors. Not shown is the control register datapath by which the 80960 processor configures and controls the stream processors.

The Cheops system has three classes of modules which can be swapped into a card cage depending on the particular application being run. The *memory/input modules* are the source and storage for data. A memory card capable of storing up to 8 GB of data acts as a main memory store for the system, and also as a high bandwidth (400 MB/s) interface to a

set of daughtercards [1]. The input daughtercard (currently under construction) acquires up to 17 video sources through one on-board digitizer/deinterlacer and two Hoover bus interfaces. A separate HIPPI daughtercard allows high speed data acquisition from other external sources.

The *processor module* routes multiple streams of data from the memory card through a full crosspoint switch which redirects them through one or more stream processors and VRAM memory banks to achieve the desired output stream. There are several stream processors available: filter processor, DCT processor, motion estimator, color space convertor, remap and composite processor, superposition processor, sequenced lookup table, and the reconfigurable state machine.

In addition to the stream processors, a general purpose RISC i960 microprocessor in the processor module handles the lower bandwidth control and configuration signals through register interfaces to each of the other modules.

The output module displays the output stream onto a variety of output devices. The O1 Output Card can support video resolutions up to 1280x1024 at 66 Hz, and a separate O2 Output Card can drive a 2048x2048 monitor at 60 Hz.

For further information on the Cheops Imaging System, please refer to *Cheops: A Reconfigurable Data-Flow System for Video Processing* by V. Michael Bove Jr. and John A. Watlington, presented in the IEEE Transactions on Circuits and Systems for Video Technology, Vol 5., No. 2, April 1995.

Appendix D: 6811 Utility Source Code

D.1 Source Code for convertor.c

```
/* convertor.c takes as an ASCII S19 formatted file as input and returns the raw
binary 6811 machine code file.
```

```
Written by Jeffrey Wong
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define BUFFER_SIZE 2
```

```
int process_line (FILE *ip, FILE* op);
```

```
void main()
{
```

```
    char filename[80];
    FILE *ip;
    FILE *op;
```

```
    char buffer[2];
```

```
    printf("Enter input filename: ");
    scanf("%s", filename);
```

```
    if ((ip = fopen(filename,"r")) == NULL)
    {
        printf("\nError accessing input file%s.\n",filename);
        exit(-1);
    }
    else printf("File %s opened successfully.\n", filename);
```

```
    printf("Enter output filename: ");
    scanf("%s", filename);
```

```
    if ((op = fopen(filename,"w")) == NULL)
    {
        printf("\nError accessing output file %s.\n",filename);
        exit(-1);
    }
    else printf("File %s opened successfully.\n",filename);
```

```
    /* begin loop of reading file here */
```

```
    while (process_line(ip, op) == 0);
```

```
    fclose(ip);
    fclose(op);
```

```
}
```

```
/* This function processes lines of an S1 formatted assembly file. Checks for
an initial S1 or S9 at the beginning of the line. It then extracts the
number of characters in the line from the next two characters using strol(),
and then reads that number of two character units. It then pries off the
```

```

    CR at the EOL and returns 0 if this was a S1 line, -1 otherwise.
*/

int process_line(FILE *ip, FILE *op)
{
    char buffer[BUFFER_SIZE];
    int counter, i;
    unsigned char *charptr;

    charptr = (char *) (malloc(sizeof(unsigned char)));

    fread(buffer, (sizeof(char)), BUFFER_SIZE, ip);

    if (memcmp(buffer, "S1", BUFFER_SIZE) == 0)
    {
        printf("Processing a S1 line\n");
        fread(buffer, (sizeof(char)), BUFFER_SIZE, ip);

        counter = (int) (strtol(buffer, (char **) NULL, 16));
        printf("Counter value extracted is %d\n", counter);

        /* parse out the address here */

        for (i = 0; i < 2; i++)
        {
            fread(buffer, (sizeof(char)), BUFFER_SIZE, ip);
        }

        printf("Processed address\n");

        /* now write out the MEAT of the line to the output file */

        for (i = 2; i < (counter - 1); i++)
        {
            fread(buffer, (sizeof(char)), BUFFER_SIZE, ip);
            printf("Processing %c%c\n", buffer[0], buffer[1]);

            *charptr = (unsigned char) (strtol(buffer, (char **) NULL, 16));
            printf("Writing %x here\n", *charptr);
            fwrite(charptr, (sizeof(unsigned char)), 1, op);
            /* insert magic code here */
        }

        /* read out the checksum; perform no operations upon it */
        fread(buffer, (sizeof(char)), BUFFER_SIZE, ip);

        /* read the CR character */
        fread(buffer, (sizeof(char)), 1, ip);

        printf("Done with the S1 line\n");
    }
    else
    {
        if (memcmp(buffer, "S9", BUFFER_SIZE) == 0)
        {
            printf("Processing a S9 line\n");
            fread(buffer, (sizeof(char)), BUFFER_SIZE, ip);
        }
    }
}

```

```

        counter = (int)(strtol(buffer, (char **)NULL, 16));
        printf("Counter value extracted is %d\n", counter);

        for (i = 0; i < counter; i++)
        {
            fread(buffer, (sizeof(char)), BUFFER_SIZE, ip);
        }
        /* read the CR character */
        fread(buffer, (sizeof(char)), 1, ip);

        printf("Done with the S9 line\n");

        return(-1);
    }
    else
    {
        printf("Error - not a S1 formatted file\n");
        printf("Aborting conversion process\n");
        return(-1);
    }
}
return(0);
}

```

D.2 Source Code for MCU EEPROM Programmer Routine

```

* BOOTLOADER PROGRAM FOR 68HC11E2 - EEPROM FILLER
*
*****
* EQUATES FOR USE WITH INDEX OFFSET = $1000
*
*
PORTD    EQU        $08
DDRD     EQU        $09
TOC1     EQU        $16      [EXTRA STORAGE (POOR STYLE)]
SPCR     EQU        $28      (FOR DWOM BIT)
BAUD     EQU        $2B
SCCR1    EQU        $2C
SCCR2    EQU        $2D
SCSR     EQU        $2E
SCDAT    EQU        $2F
BPROT    EQU        $35
PPROG    EQU        $3B
TEST1    EQU        $3E
CONFIG   EQU        $3F
*
* MORE EQUATES
*
EEPSTR    EQU        $F800      START OF EEPROM
EEPEND    EQU        $FFFF      END OF EEPROM
*
*****
* THIS BOOTSTRAP PROGRAM ALLOWS THE USER TO
* DOWNLOAD A PROGRAM OF 0 TO 0x800 BYTES INTO THE HC11 EEPROM, AS SPECIFIED BY
* THE VALUE IN THE LINE WITH THE COMMENT PUT_SIZE_OF_PROGRAM_IN_BYTES_HERE.
* EACH BYTE OF THE PROGRAM IS RECEIVED BY THE SCI.
* THE MCU IS ASSUMED TO HAVE A 4.9152 MHZ XTAL, NOT A 8 MHZ XTAL.
* THIS ALLOWS THE SCI TO HAVE A 38.4K BAUD RATE.
* THE FIRST BYTE ($FF) CONFIRMS THIS BAUD RATE, OR ELSE DROPS TO 9600 BAUD.
* THEN THE PROGRAM IS DOWNLOADED STARTING WITH
* THE $0000 BYTE AND WORKING UP TOWARD THE TOTAL NUMBER OF BYTES SPECIFIED.
*

```

* THE TRANSMITTER WILL ECHO THE BYTE WRITTEN INTO EEPROM WHEN IT HAS FINISHED
 * THE EEPROM PROGRAMMING ROUTINE. DO NOT SEND ANOTHER CHARACTER UNTIL YOU
 * HAVE RECEIVED THE PREVIOUS ONE.

- Jeff Wong 7/9/96

* Note: Parts of this code have been taken from Motorola code sources.
 * Credit is given for the code.

```

BEGIN EQU *
* INIT STACK
    LDS    #$00FF
* INIT X REG FOR INDEXED ACCESS TO REGISTERS
    LDX    #$1000
* PUT PORT D IN WIRE OR MODE
    BSET   SPCR,X $20
* INIT SCI AND RESTART BAUD DIVIDER CHAIN
    LDAA   #$81* DIV BY 1228800 by 16*2
    STAA   BAUD,X
* RECEIVER & TRANSMITTER ENABLED
    LDAA   #$0C
    STAA   SCCR2,X
*****
* clear reg for eeprom prog and erase
    CLR    BPROT,X
* SEND BREAK TO SIGNAL START OF DOWNLOAD
    BSET   SCCR2,X $01
* CLEAR BREAK AS SOON AS START BIT IS DETECTED
    BRSET  PORTD,X $01 *
    BCLR   SCCR2,X $01 * CLEAR BREAK
* WAIT FOR FIRST CHARACTER (USERS SEND $FF)
    BRCLR  SCSR,X $20 * * WAIT FOR RDRF
    LDAA   SCDAT,X * READ DATA
* IF DATA = $FF, THEN /(16*2) IS CORRECT BAUD
    CMPA   #$FF
    BEQ    BAUDOK
* ELSE CHANGE TO /(16 * 8) TO GET 9600 BAUD
    BSET   BAUD,X $03
BAUDOK EQU *
* THEN DOWNLOAD PROGRAM
    LDY    #$F800 * INIT POINTER
* READ IN PROGRAM AND PUT INTO RAM
BK2 EQU *
    BRCLR  SCSR,X $20 * WAIT FOR RDRF
    LDAA   SCDAT,X
    LDAB   #$16 * EELAT=1, BYTE ERASE
    STAB   PPROG,X
    STAB   0,Y * ACCESS BYTE POINTED TO BY Y
    LDAB   #$17 * TURN ON EEPGM
    STAB   PPROG,X
    BSR    DELAY10 * RELATIVE ADDRESSING IS LOCATION INDEPENDENT
    CLR    PPROG,X * TURN OFF EEPGM AND EELAT
*****
* THIS CODE PROGRAMS THE BYTE WHOSE ADDRESS IS IN
* REGISTER Y WITH THE VALUE PASSED IN REGISTER A
*****
BYTEPRG LDAB   #$02 * EELAT=1, EEPGM=0
    STAB   PPROG,X
    STAA   0,Y * WRITE DATA INTO ADDRESS POINTED TO BY Y
    LDAB   #$03 * EEPGM=1
    STAB   PPROG,X
    BSR    DELAY10 * RELATIVE ADDRESSING IS POSITION INDEPENDENT
    CLR    PPROG,X * EELAT=EEPGM=0
    STAA   SCDAT,X * HANDSHAKE - BY RETURNING BYTE HERE

```

```

        INY
        CPY #$0100      * PUT_SIZE_OF_PROGRAM_IN_BYTES_HERE
        BNE BK2
        BRA *           * INFINITE SELF LOOP TO ENDS EXAMPLE
*****
* THIS SUBROUTINE DELAYS THE MCU FOR 10 MS
*****
DELAY10 PSHX           * MAKE REENTRANT AND SAVE REGISTERS
        LDX      $0D05 * CONSTANT FOR 10MS AT 2 MHZ E-CLOCK
LOOP10  DEX
        BNE      LOOP10 * RELATIVE ADDRESSING IS POSITION INDEPENDENT
        PULX           * RESTORE REGISTERS
        RTS
FILLER0 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
FILLER1 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
FILLER2 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
FILLER3 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
FILLER4 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
FILLER5 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE

```

```

NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLER6 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLER7 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLER8 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLER9 NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLERA NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLERB NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP

```

```

NOP
NOP
NOP
NOP
NOP
FILLERC NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLERD NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLERE NOP* NECESSARY TO MAKE BOOTLOADER THE RIGHT SIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
FILLERF NOP* GUESS WHAT THE COMMENT SHOULD BE OVER HERE
NOP
NOP      * IF YOU LEFT WHEN THE CREDITS BEGAN, YOU'D BE HOME NOW

```

Bibliography

- [1] Bove, V. Michael Jr., and John A. Watlington. "Cheops: A Reconfigurable Data-Flow System for Video Processing." *IEEE Transactions on Circuits and Systems for Video Processing*, April 1995, pp. 140-149.
- [2] Nossokoff, Mark and Gene Freeman. "SCSI R.I.P. - NOT! The Case for FAST-20 SCSI." *Computer Technology Review*, April 1995. Also available at http://www.symbios.com/articles/ctr_495.htm.
- [3] Goldie, John. "An Introduction to the Differential SCSI Interface." *National Semiconductor Application Note 904*, August 1993. Available online from <http://www.national.com>.
- [4] Balakrishnan, R. V. "Reducing Noise on Microcomputer Buses." *National Semiconductor Application Note 337*, July 1986. Available online from <http://www.national.com>.
- [5] Small Computer System Interface - 2, X3.131-199x, ANSI.
- [6] Kunzman, Adam J. and Alan T. Wetzel. "1394 High Performance Serial Bus: The Digital Interface for ATV." *IEEE Transactions on Consumer Electronics*, August 1995, Vol. 14, No. 13, pp 893-900.
- [7] Teener, Michael. "New Technology in the IEEE P1394 Serial Bus - Making it Fast, Cheap and Easy to Use." Presented to the Hot Interconnects Symposium '93 at Stanford University, August 6, 1993.
- [8] Tving, Ivan. "Multiprocessor interconnection using SCI." Masters Thesis, DTH ID-E 579, 1994.
- [9] James, David and David Gustavson. "Draft Proposals for Real-Time Transactions on SCI." SCI/RT-1995.
- [10] Gustavson, David B. Email correspondence. Aug 1996.

- [11] Bertsekas, Dimitri P. and Robert Gallager. *Data Networks*. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1992.
- [12] Borrill, Paul and John Theus. "An Advanced Communication Protocol for the Proposed IEEE 896 Futurebus." *IEEE Micro*. Vol 4, No. 4, August 1984, pp 42-56.
- [13] DiGiacomo, Joseph Di (ed). *Digital Bus Handbook*. McGraw Hill Publishing Company. New York, New York, 1990.
- [14] Gustavson, David B. "Computer Buses - A Tutorial." *IEEE Micro*. Vol 4, No. 4, August 1984, pp 7-22.
- [15] Hosseini-Khayat, Saied and Andreas D. Bovopoulos. "A Simple and Efficient Bus Management Scheme that Supports Continuous Streams." *ACM Transactions on Computer Systems*, Vol 13, No. 2, May 1995, pp.122-140.
- [16] *M68HC11 E Series: Technical Data*. Motorola Semiconductor, 1995.
- [17] Taub, D. M. "Arbitration and Control Acquisition in the Proposed IEEE 896 Futurebus." *IEEE Micro*. Vol 4, No. 4, August 1984, pp 7-22.
- [18] "RS-422 and RS-485 Application Note." B&B Electronics Manufacturing Company. Available at <http://www.bb-elec.com/appnote/appnote1.html>.