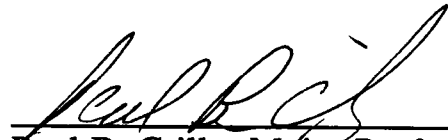
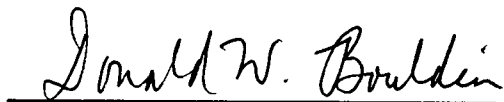


To the Graduate Council:

I am submitting herewithin a thesis written by Kenneth Franklin Baker, Junior entitled "The Design of a Priority Event Monitor for an Interrupt-Driven Computer." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Electrical Engineering.

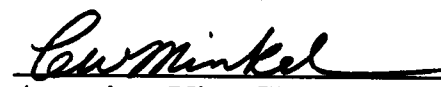

Paul B. Crilly, Major Professor

We have read this thesis
and recommend its acceptance:


Donald W. Bouldin


Robert E. Bodenheimer

Accepted by the Council:


Associate Vice Chancellor
and Dean of The Graduate School

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under the rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgements of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature K F Baker J.

Date April 9, 1992

THE DESIGN OF A PRIORITY EVENT MONITOR FOR AN INTERRUPT-DRIVEN COMPUTER

A Thesis
Presented for the
Master of Science
Degree

The University of Tennessee, Knoxville

Kenneth Franklin Baker, Junior

May 1992

DEDICATION

This thesis is dedicated to my wife, Maureen Ryan Baker, for her love, care and patience during the work involved.

ACKNOWLEDGEMENTS

I would like to thank Dr. Paul B. Crilly, Dr. Robert E. Bodenheimer, and Dr. Donald W. Bouldin, who served as my major professor and committee members respectively. Thanks go to Dr. Crilly for his guidance in the design work and to Dr. Bodenheimer for the use of laboratory space and equipment. Thanks go also to Dr. Bodenheimer and Dr. Bouldin for their excellent teaching during my undergraduate and graduate degree courses.

I am grateful to International Business Machines for funding my research and design work and special thanks to Lloyd J. Conard at IBM, Charlotte, North Carolina. Mr. Conard's patience for my many questions and his excellent guidance as my primary contact at IBM are appreciated greatly.

This work was done in the Digital Systems Laboratory at the University of Tennessee, Knoxville, Tennessee for International Business Machines.

ABSTRACT

An interrupt-driven microprocessor operates using priority levels to indicate which of multiple interrupt sources is currently being serviced, with a wait state to indicate that no interrupts are active. By monitoring and timestamping priority level and wait state activity, an analysis tool for an interrupt-driven microprocessor can be implemented. This thesis describes the design of such a tool, the Priority Event Monitor, for the SOLO microprocessor, proprietary to IBM.

The Priority Event Monitor consists of a PS/2 Microchannel computer, hardware for observing priority activity on the SOLO processor, and a SCSI bus link between the PS/2 and the hardware. The Priority Event Monitor digital architecture and the decisions leading to that architecture are covered, and the hardware architecture is shown and described.

TABLE OF CONTENTS

	Page
Chapter 1 Introduction	1
1.1 The IBM 4702 and SOLO Microprocessor.....	2
1.2 Events.....	3
1.3 Timestamps.....	4
1.4 Event Records.....	5
1.5 Thesis Guide.....	6
Chapter 2 IBM 4702 Operation and Event Signatures.....	8
2.1 IBM 4702 Operations.....	8
2.2 SOLO Microprocessor Signal Definitions.....	9
2.3 Event Signatures.....	11
2.4 Project Specifications.....	13
Chapter 3 Priority Event Monitor Architecture Considerations.....	17
3.1 Priority Event Monitor Overview.....	17
3.2 Priority Event Monitor Architecture.....	18
3.3 Event Record Format and System Throughput.....	19
3.4 External Interface Considerations.....	22
3.4.1 Physical Considerations.....	23
3.4.2 Comparisons of Available Buses.....	23
3.4.3 SCSI Logical Interface.....	24
3.4.4 Logical SCSI Interface Simplification.....	26
3.5 Personal Computer Control.....	27

Chapter 4	Monitor Hardware Architecture.....	30
4.1	Monitor Architecture.....	30
4.2	FIFO Buffers.....	32
4.3	Event Capture.....	36
4.3.1	Wait Event Capture.....	38
4.3.2	Level Event Capture.....	42
4.4	Timestamp Module and Timer Overflow Detection.....	48
4.5	Write Controller.....	51
4.5.1	Write Controller Overview.....	52
4.5.2	Write Controller Implementation.....	55
4.5.3	Write Controller State Machine.....	57
4.6	Small Computer Systems Interface.....	65
4.6.1	SCSI Physical Implementation.....	66
4.6.2	SCSI Data Packet Size Derivation.....	68
4.7	DMA Controller.....	73
4.8	Microprocessor.....	84
Chapter 5	Summary and Conclusions.....	89
5.1	Results of Present Implementation.....	89
5.2	Suggested Priority Event Monitor Improvements.....	91
5.2.1	Architectural Change.....	91
5.2.2	Monitor Hardware Changes.....	93
5.3	Suggested Implementation Changes.....	95

References.....	97
Appendices.....	99
Appendix A Original Project Specifications.....	100
Appendix B Timer Field Length Derivation.....	109
Appendix C PS/2 Communication and Control Software Listing.....	116
Appendix D PAL Specification Listing.....	161
Appendix E Monitor Hardware Schematics.....	211
Vita.....	218

LIST OF FIGURES

	Page
Figure 3.1: Partitions of the Priority Event Monitor.....	18
Figure 3.2: Priority Event Monitor Architecture.....	19
Figure 3.3: Representation of the Event Record.....	20
Figure 3.4: Configuration of the Comment Field.....	21
Figure 4.1: Monitor Architecture.....	31
Figure 4.2: Block Diagram for the FIFO Buffer.....	33
Figure 4.3: Event Capture Module Block Diagram.....	37
Figure 4.4: Wait Capture Block Diagram.....	39
Figure 4.5: Wait Machine State Diagram.....	41
Figure 4.6: Block Diagram for Level Capture.....	42
Figure 4.7: Event Capture Module - Level Capture.....	44
Figure 4.8: LPL State Diagram.....	45
Figure 4.9: CPL Machine State Diagram.....	46
Figure 4.10: LC State Diagram.....	47
Figure 4.11: Block Diagram of the Timestamp Module.....	49
Figure 4.12: Write Controller Block Diagram.....	54
Figure 4.13: Timing of the WR_EN signal.....	58
Figure 4.14: Write Controller State Diagram.....	63

Figure 4.15: Line Termination for SCSI.....	66
Figure 4.16: Block Diagram for the SCSI Interface.....	67
Figure 4.17: Overall View of the DMA Controller.....	74
Figure 4.18: DMA Initiator Write Timing for the 53C80.....	75
Figure 4.19: Read Timing for the Am7200-80 FIFO.....	77
Figure 4.20: Block Diagram for the DMA Controller.....	79
Figure 4.21: State Diagram for the DMA Controller State Machine....	81
Figure 5.1: Improved Priority Event Monitor Architecture.....	92
Figure 5.2: Improved Monitor Architecture.....	94
Figure B-1: Graphical Results of Equation B-4.....	114
Figure B-2: Byte Rate vs. Event Rate with Emphasis on Operating Area.....	114

LIST OF TABLES

	Page
Table 2.1: Definition of SOLO Control Bus Signals.....	10
Table 2.2: SOLO Control Signal Combinations for Waits and Levels...	12
Table 3.1: Command Byte Assignments.....	20
Table 4.1: Timing Values for Figure 4.18.....	76
Table 4.2: Timing Values for Figure 4.19.....	77
Table B-1: Byte Rate vs. Event Rate.....	113

CHAPTER 1

INTRODUCTION

Interrupt-driven microprocessors operate with priority levels assigned to each interrupt source; the priority levels are used to resolve disputes between interrupts. When multiple interrupts are active then the highest priority interrupt has precedence and its service routine assumes control of the processor. Lower priority interrupts are serviced as higher priority service routines finish. If no interrupts are active then a wait state is entered at the present priority level, idling the microprocessor until the next interrupt. Once an interrupt occurs, the microprocessor exits the wait state at the present priority level, then, if needed, changes priority level.

Software to monitor priority level activity would require each interrupt service routine to recognize and store the fact that its processing was beginning and ending. The service routine would also be responsible for noting when these changes occurred. Besides adding extensive software processing overhead, this method of monitoring would require a large amount of main processor memory, useful for other tasks.

If the priority level and wait state information for the processor is available on an external bus, this information can be monitored and analyzed. Monitoring the priority levels externally removes the software overhead and the memory requirements from the system being monitored,

thus allowing unobtrusive observance of processor activity. The unobtrusive monitoring of priority levels, or more specifically the changes in levels, leads to implementation of a high-level microprocessor analysis tool. This tool can monitor changes in wait states and priority levels, give a time relationship to these changes, and store this information for analysis. The purpose of the work described here is the design of a high-level microprocessor analysis tool for the IBM 4702 computer.

1.1 The IBM 4702 and SOLO Microprocessor

The IBM 4702 computer, used in the banking industry, is generally used by bank branches and small main offices for control of local teller information, automatic teller machines, communication with a central bank host, and other functions. These computers use a proprietary interrupt-driven microprocessor called the SOLO microprocessor, implemented to allow access to the priority level and wait information through an external bus.

The SOLO microprocessor operates with eight priority levels, numbered zero through seven, and contains one wait state. The analysis tool, called the Priority Event Monitor, is to detect changes in operating priority levels for the SOLO microprocessor, detect entry into the wait state, and detect an exit from the wait state. Wait state activity and changes

in priority level, called Events, are given a timestamp value to allow proper time relationships to be established. The Events and timestamps are stored into a file on an IBM PS/2 personal computer where the information can be analyzed.

1.2 Events

When monitoring the SOLO microprocessor bus, changes in the priority levels are important. Any change in the operating priority level of the SOLO microprocessor or any wait state activity is called an Event. It is the Events which are logged and given a timestamp value by the Priority Event Monitor. With the Event and its timestamp, a reconstruction of all priority activity on the SOLO microprocessor and the 4702 series computer can be performed. This reconstruction allows a high level, unobtrusive view of operations inside the computer without additional program overhead.

Three Events are needed to reconstruct an event timeline of priority action on the 4702: a Priority Level Change, a Wait Entry, and a Wait Exit. The Priority Level Change is used to record changes in the active priority for any of the eight priority levels. The Wait Entry and Exit are used to record a change active priority levels and the wait state. With these three Events ordered reconstruction of priority activity can occur.

1.3 Timestamps

The three Events, the Priority Level Change, the Wait Entry, and the Wait Exit, contain enough information to order events within the 4702, but are not enough to reconstruct event relationships; a timing reference of some sort is needed to relate events in time to one another. By joining an Event and a timestamp, reconstruction of the order and timing relationship of Events is possible.

Three Events have been discussed so far, but the need for a fourth Event exists. The timer needed to form the timestamp is implemented in hardware, and an large hardware timer is impractical, due to the amount of hardware required. To overcome this fact, and to allow formation of an infinite time relationship, a fourth event is needed, the Timer Overflow. This Event will occur when the hardware timer counts from its largest value to zero, thus the Timer Overflow acts as higher order bits for the timestamp counter.

The four Events for timeline reconstruction, Priority Level Change, Wait Entry, Wait Exit and Timer Overflows, with the associated timestamp, contain the information needed to form an order and timing relationship for activity on the 4702 computers.

1.4 Event Records

The four Events, Priority Level Change, Wait Entry, Wait Exit, and Timer Overflow, with an associated timestamp, allow for reconstruction of the order and time relationship of priority changes and wait state activity. The grouping of an Event, the timestamp for the Event, and any other necessary information, is called an Event Record. It is the Event Records that are handled and analyzed to recreate the timeline of activity for the SOLO processor.

Previous IBM work on an earlier Event monitor helped form the content of the Event Records (see the specifications in Section 2.4 for more complete information). Event Records consist of three sections, an Event indicator, a timestamp value and level information in three information fields, the Command field, Time field, and Comment field respectively. The Command field distinguishes different Events for the Event Records. The Time field indicates when the Event occurred, and the Comment field is used to indicate the priority before and after a Priority Level Change. The Comment field for Timer Overflows and for Wait Events is present but unused (zero in value). The Event Record, with the Command field, the Time field, and the Comment field is the basic unit for storage and transfer within the Priority Event Monitor.

1.5 Thesis Guide

In the pages that follow, the architecture and design of the Priority Event Monitor is given. Particular attention is given to the hardware architecture and design used for logging the Events and forming Event Records.

Chapter 2 covers operation of the IBM 4702 as it relates to Event identification, and covers signal definition and signatures for the Priority Level Change, Wait Entry, and Wait Exit Events. Chapter 2 ends with a summary of pertinent IBM specifications regarding the hardware design and Event Record throughput rates.

Chapter 3 covers an overview of the Priority Event Monitor and its architecture. The use of the Priority Event Monitor is explored in greater detail and conclusions for the final architecture drawn. This chapter covers the structure used for Event storage, the Event Record, and discusses Event rates and user control of Event captures.

Chapter 4 delves into the architecture for the Monitor, the external hardware used to interface to the 4702 and capture Events. Design explanations are given for the modules that capture the data from the 4702 and form Event Records. Local control of the Monitor is explained as well as transfer and storage of Event Records.

Chapter 5 summarizes the design of the Priority Event Monitor and discusses results. Possible alternatives and improvements for both the Priority Event Monitor and the Monitor hardware are also shown.

The Appendices list the original specifications for the project, as supplied by International Business Machines, drawings and PAL hardware description files for the current implementation and program listings for control programs.

CHAPTER 2

IBM 4702 OPERATION AND EVENT SIGNATURES

This chapter covers general operation of the IBM 4702 as it relates to level and wait state Events and covers indicators for those Events. In the pages that follow, actions are defined describing priority level and wait operations on the IBM 4702. Also defined are the 4702 processor signals and the signatures for a Priority Level Change, Wait Entry, and Wait Exit in terms of those signals. Following Event signature definition, project specifications are summarized.

2.1 IBM 4702 Operations

Priority level and wait state operations of the 4702 computer are explained in this section, to allow the reader an idea of the process taking place inside the 4702 and to define several important terms for later use in of Event signature description and hardware design. When the 4702 processor is changing priority levels, two steps are taken. First the processor saves control data for the Last Priority Level (LPL) to a specific set of registers for use when the interrupt routine is active again. This allows the interrupted service routine to begin processing again in the exact state before interruption. Second, the processor retrieves control data for

the new priority level, called the Current Priority Level (CPL), from its specific set of registers. It is during the register storage and retrieval that the level information needed for the Priority Level Change is available. A segment of the 4702 address bus indicates the set of registers being accessed and therefore the priority level being stored or retrieved. By capturing the address bus segment the values for LPL and CPL is determined. At CPL register retrieval the Priority Level Change is complete.

Wait state operations, entry and exit, on the IBM 4702 are intimately tied to the Priority Level Changes. When a current priority level service routine finishes operation and no other interrupts need service, the 4702 enters a wait state at the current priority level, with the 4702 remaining in the wait state until the next interrupt occurs. When this interrupt occurs, the processor exits the wait state at the same level at which it entered, then executes a Priority Level Change if needed. The processor enters and exits the wait state at the same priority level because the current priority level control information is not stored until the Priority Level Change.

2.2 SOLO Microprocessor Signal Definitions

There are a total of nine signals needed for defining Event signatures. Six of the signals are available on the SOLO control bus and three are on the SOLO address bus. The data bus is not used in

determining wait or priority level Events. The control bus signals determine when the Event occurs and the address bus contains the level information for Priority Level Changes. The six control bus signals and their definitions are given in Table 2.1.

Table 2.1: Definition of SOLO Control Bus Signals. These signals are important for Wait and Level captures. Note that the “/” indicates active low signals.

<u>Signal Name</u>	<u>Basic Definition</u>
/IFWT	Instruction Fetch and Wait Indicator
/IRPT	Interrupt Active
/IOTAG	Input/Output Indicator
/MSEL	Memory Select
/RSEL	Register Select
/WRT	Memory or I/O Write

The “/” before each signal designates the signal as active low. As all signals to be discussed have active low polarity the active low indicator will not be used in discussions. IFWT is active when either an Instruction Fetch is occurring or the 4702 is in the wait state. IRPT is active when any interrupt occurs. IOTAG is active when any input or output is occurring. MSEL indicates that main memory is being accessed and RSEL indicates that register space is being accessed. WRT is active when writing and inactive when reading memory, registers, or input and output.

The address signals, containing the level information for Priority Level Changes are /ADDR27, /ADDR28, and /ADDR29. These signals contain the three bit value indicating the level information. During a level change, the information for the last priority level (LPL) and the current priority level (CPL), with the eight possible values encoded, are multiplexed on these three address lines. ADDR27 is the most and ADDR29 is the least significant bit.

2.3 Event Signatures

Three Events are needed to build the timeline of priority action on the IBM 4702, Priority Level Changes, Wait Entry, and Wait Exit, and these Events are related to specific signatures defined by the control bus of the IBM 4702. The wait state is defined by the signals IFWT, IRPT, IOTAG and MSEL. When these four signals are at the correct value, as defined by table 2.2, the processor for the IBM 4702 is in a wait state. The Wait Entry Event is defined by the transition from active processing to a wait state. Further more, once the 4702 has entered a wait state, IFWT is the only signal guaranteed to be valid; the other signals, IRPT, IOTAG and MSEL will float. This condition must be taken into account in order to avoid false Wait Exits and Entries.

The Wait Exit Event is defined by a similar sequence as the Wait Entry Event. First the processor must be in a wait state, as described in the proceeding paragraph. Second the signal IFWT must become inactive; this is the only indicator that the processor is no longer in a wait state, as the values for the other indicator signals, IRPT, IOTAG and MSEL are not guaranteed.

Table 2.2: SOLO Control Signal Combinations for Waits and Levels. These combinations indicate wait state activity and Last Priority Level and Current Priority Level availability. Note that the information given indicates signal activity, not signal polarity.

<u>Information</u>	<u>Signature</u>
Wait State Active	IFWT*/IRPT*/IOTAG*/MSEL
LPL Available	IFWT*IRPT*RSEL*WRT
CPL Available	IFWT*IRPT*RSEL*/WRT

The Priority Level Change Event is also defined by a sequence of 4702 actions, given in detail in Section 2.1 and outlined here. The 4702 first stores the LPL control information into register space, then retrieves the CPL information from register space. A Priority Level Change is indicated by the above sequence, and needs information from both the LPL and CPL phases. A Priority Level Change is complete when CPL information has been captured from the 4702 address bus.

The information given in Table 2.2 summarizes the Event signatures for the Priority Level Change, Wait Entry, and Wait Exit. A Wait Entry record is formed whenever a transition to a wait state occurs and a Wait Exit record is formed whenever a transition from wait state occurs. A Priority Level Change, though is somewhat different. This Event depends on two pieces of information, the LPL and CPL values. A level change is not complete until CPL values are available, so LPL information must be stored until the CPL value is ready.

2.4 Project Specifications

This section covers an abbreviated list of project specifications, as written and discussed with IBM. The full initial written specifications are given in Appendix A. The specification summary given here is broken into 6 sections: General Overview, Timestamp, Event Capture and Control, Event Recording Rates, PS/2 Control, and Event Record Formats. The specification contains information for the project description given in the following chapters.

General Overview

Objective - Design, build, code, and debug a PS/2 Based 4702 real-time performance monitor.

General Hardware Requirements - Record and timestamp selective 4702 processing Events. This includes Priority Level Changes, Wait State Entry, and Wait State Exit.

There will be 3 physical entities: A Redrive Card, already designed at IBM, for boosting the drive capability of the 4702 signals. Monitor hardware for observing the 4702 signals and recognizing Events, and a PS/2 with a Microchannel adapter card.

General Software Requirements - There are two major pieces of PS/2 software: A Measurement Setup and Control program, and a Post Measurement Reduction and Converter program.

The Measurement Setup and Control program is responsible for real time control of the Monitor hardware from the PS/2 and for storage of Event Records from the Monitor hardware.

The Post Measurement Reduction and Converter program must take the stored Event Records and remap timer functions and convert the data format to be compatible with the older 4702 Performance Monitor format.

Timestamp

Timestamp Resolution - The Timestamp timer must be driven by a local oscillator and must be able to handle the 1 microsecond timestamp resolution.

Timestamp Length - The timestamp must be 1 or more bytes. An 8 bit length is minimum.

Timer Overflows - All Timer Overflow Events must be recorded.

Timer Structure - The timestamp timer can either be reset with each recorded event or can be free running.

Event Capture and Control

Event Record Structure - The Event Record will have 1 byte indicating the Event type (Command Byte), 1 or more bytes for the Timestamp, and 1 byte for details (Comment Byte). Therefore, an Event Record will consist of at least 3 bytes.

PS/2 Control - The Monitor hardware must be able to react to asynchronous commands from the PS/2.

Event Recording Rates

Peak Rate - The Monitor hardware must be able to record Event Records at a 1 μ second rate for up to 32 Events, or in other words, the hardware must be able to sustain a capture rate of 32 Event Records in 32 μ seconds.

Average Rate - The Monitor hardware must be able to sustain indefinitely a rate of 1 Event Record every 32 μ seconds.

Record Rate - It is possible to have two Event occur in the same microsecond. Both these Events should contain the same Timestamp and be ordered by priority.

Upload Rate - The Monitor hardware must be able to sustain the above rates while uploading captured Event Records to the PS/2.

PS/2 Control

Event Record Upload - The PS/2 Microchannel interface must be able to react to asynchronous commands from the Monitor hardware, indicating that Event Records are ready to be uploaded.

Event Capture Control - The PS/2 should be able to control whether Wait and Priority Level Change Events are captured. Timer Overflows are always captured.

Event Record Format

The following information describes the Event Record format used for the Priority Event Monitor. This information follows a previous Event Record format used at IBM, with only one change, storage of Wait Exit Events. Wait Exit Events are remapped using the Post Measurements Reduction and Converter Program.

Command Byte 00h - A Basic Timer Overflow Record. The Timestamp value and the Comment byte should be 0h.

Command Byte 02h - A Priority Level Changed Event. The Timestamp value is included, and the Comment byte should indicate the last and current priority level.

Command Byte 04h - Wait State Entry Event. The Timestamp is included, and the Comment byte should be 0h.

Command Byte 06h - Wait State Exit Event. The timestamp is included, and the Comment byte is 0h. This Command byte type is the only difference between the older IBM format and the format used for the present design. The older format used the 02h Command byte to represent the Wait Exit as well as the Priority Level Changes. Forming a separate Command byte for this Event reduces the needed hardware for the system.

The Specifications above are abridged from the full specification list given by IBM, with the full list of specifications given in Appendix A. The abridged information above covers the major specifications needed to continue with a thorough discussion of the project and its design.

CHAPTER 3

PRIORITY EVENT MONITOR ARCHITECTURE CONSIDERATIONS

This chapter discusses the Priority Event Monitor with the system level partitions shown. Event Record formats and byte throughput considerations are discussed as well as the interface between the PS/2 and the Monitor. A discussion of PS/2 control of the Priority Event Monitor is also included.

3.1 Priority Event Monitor Overview

The purpose of the Priority Event Monitor is the capture, storage and analysis of level and wait information from the IBM 4702 computers. To perform these functions, the Priority Event Monitor must observe the 4702 external busses and recognize Events. Then it must apply a timestamp and form an Event Record with the Event indicator, the timestamp, and level information. The Event Records must be transferred to a DOS file on a PS/2 personal computer, where processing of level and wait information can occur. Furthermore, the personal computer must be able to control whether priority level Events or wait Events or both are captured. With these functions in mind, two partitions exist for the Priority Event Monitor and are shown in Figure 3.1.

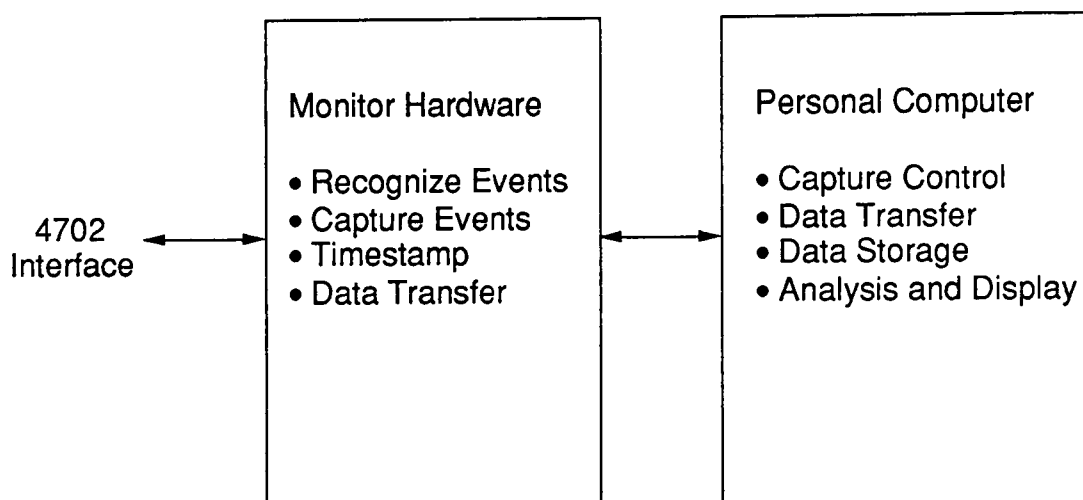


Figure 3.1: Partitions of the Priority Event Monitor.

Hardware is needed to recognize Events, apply a timestamp and transfer the Event Records to the PS/2 personal computer. The Personal Computer is responsible for receipt, storage and analysis of Event Records as well as capture control.

3.2 Priority Event Monitor Architecture

The Priority Event Monitor, represented in Figure 3.2, consists of a PS/2 personal computer, an external hardware Monitor, connection from the 4702 to the Monitor and connection between the Monitor and the PS/2.

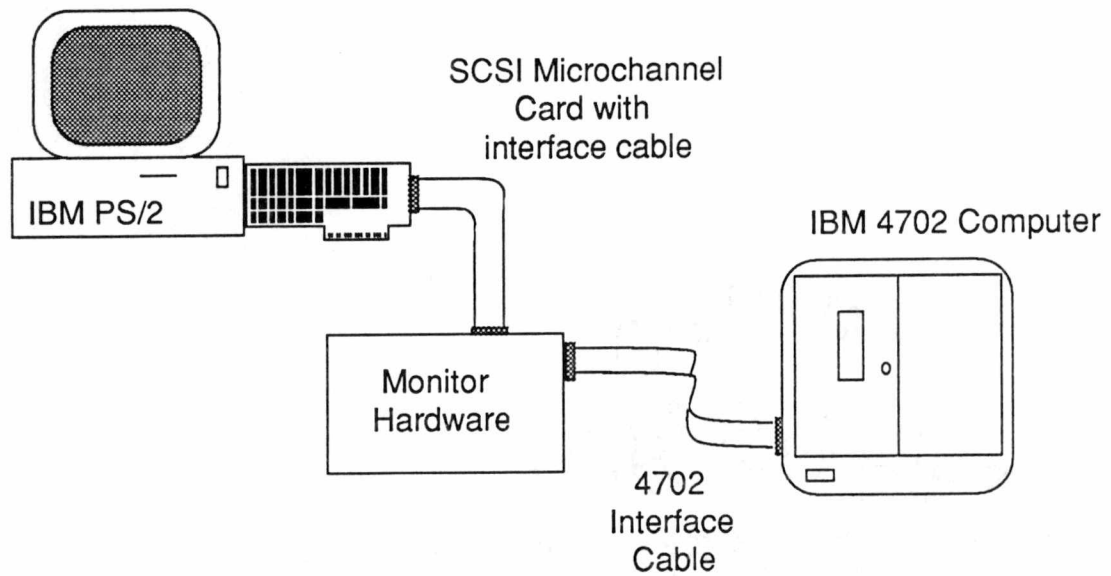


Figure 3.2: Priority Event Monitor Architecture.

The PS/2 is used for control of the Monitor and for data storage. The Monitor is used for the capture of events and formation of Event Records. The link between the 4702 and the Monitor provides signals needed for Event capture, and the link between the PS/2 and the external hardware allows control and communication between the two.

3.3 Event Record Format and System Throughput

As given in Section 1.4, Event Records are a convenient way to represent an Event and the information associated with the Event. The Event Record, following the previous IBM format, consists of three fields

with the first field holding an Event indicator, the second field holding the timestamp value, and the third field holding priority level information. The fields are given the representative names of Command field, Time field, and Comment field, respectively, as shown in Figure 3.3.

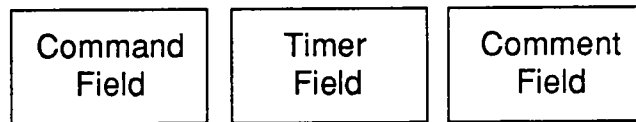


Figure 3.3: Representation of the Event Record.

The length of the Command Field is set at 1 byte, with Table 3-1 showing the Command byte assignments for the 4 Events. The Command byte values, with the exception of the Wait Exit, are compatible with the previous IBM format.

Table 3-1: Command Byte Assignments.

<u>Command Byte Value</u>	<u>Represented Event</u>
0	Timer Overflow
2	Priority Level Change
4	Wait Entry
6	Wait Exit

The Comment field has to hold CPL and LPL information which are encoded values of three bits each. With a total of six bits each, the Comment field is assigned a length of one byte, also allowing for ease of analysis and compatibility with older Event Record formats. The configuration for the Comment field, following the previous IBM format, is shown in Figure 3.4.

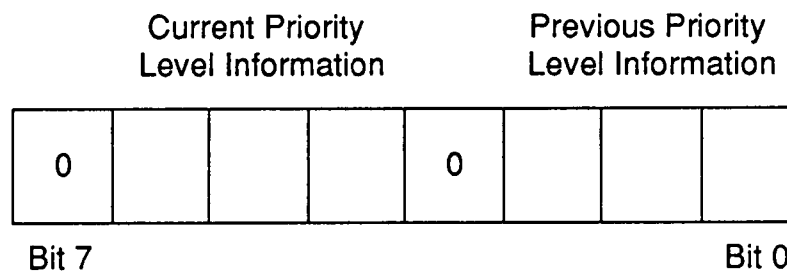


Figure 3.4: Configuration of the Comment Field.

The Time field lacks an assigned length. The timestamp timer is implemented as hardware, and three lengths for the hardware timer are considered: 1, 2, and 3 bytes. Timer lengths were all considered on the byte boundary to retain an uniform structure within the Event Records. The older IBM format uses the 1 byte timer length and lengths of 16 and 24 bits considered to reduce Timer Overflows.

The criteria for judging the length of the timer was reduction the total number of bytes needed each second, termed the byte rate. By reducing the byte rate, transmission time and data storage needs are

reduced. To reduce the byte rate, an equation for the byte rate is derived and the effect of various parameters considered. The derivation for the byte rate equation, and the results for the byte rate analysis is done in Appendix B.

The length of the Timer field is derived in Appendix B as 2 bytes. With the determination of the Time field length, the Event Record specification is now complete. The Command and Comment fields have lengths of 1 byte each, and the Time field has a length of 2 bytes. This gives the Event Record an overall length of 4 bytes. Referring to maximum specification of the Event Rate of 31250 Events/second, given in Section 2.4, the maximum byte rate for the system including overhead for Timer Overflows is 125061 bytes/second.

3.4 External Interface Considerations

The system level partition of the Priority Event Monitor is given in Section 3.2, but the link between the 4702 and the external hardware is unspecified. The communication between the PS/2 and the Monitor and the several issues involving the communication will now be discussed. The decision of communication protocols is decided by considering the bandwidth needed for data transfer to the PS/2 from the Monitor and by considering ease of implementation.

3.4.1 Physical Considerations

Two major issues are used to decide the choice of communication protocol between the PS/2 and the Monitor. These are bandwidth and the ease of implementation. Bandwidth is paramount, because data must be transferred to the PS/2 without data loss or overrun. The maximum byte rate for the Monitor was calculated in Section 3.3 to be 125,061 bytes/second. To account for software and communication overhead, a design factor of 10 was used to arrive at a figure for communication bandwidth. The minimum design bandwidth for communication between the Monitor and the PS/2 is therefore set at 1.25 Mbytes/second.

3.4.2 Comparisons of Available Buses

Several busses were considered for use at the communication link between the PS/2 and the Monitor. The two major busses considered were the IEEE-488 and the Small Computer Systems Interface (SCSI), with the major considerations being ease of implementation and the bus bandwidth. IEEE-488 is easy to implement due to its primary purpose as an instrument bus, but it was eliminated because its maximum bandwidth was 500 Kbytes/second, about half of the design bandwidth [6].

SCSI, on the other hand, has current bandwidths for synchronous transfers of 4 Mbytes/second and 1.5 Mbytes/second for asynchronous transfers [1, 2, 4]. This bandwidth is more than enough to accomplish the task needed. Furthermore, many boards exist to interface a SCSI bus to the PS/2 Microchannel bus, and these boards have interface software written to accomplish major communication protocol requirements. Also, SCSI bus interface chips needed for the Monitor are available through multiple vendors. The SCSI bus has a defined protocol for commands, data, and status, as well as other features. [1, 3]

3.4.3 SCSI Logical Interface

The logical interface of the SCSI bus is examined here. The SCSI Bus has phases of operation which are defined by the current settings of the bits available on the SCSI data bus. The phases are set by a control bus and the data are transferred on the data bus. Up to eight entities can exist on the SCSI bus, with communications existing between two and only two parties at one time [1].

The communicator transmitting is the Initiator, and the communicator receiving is the Target. The Initiator is responsible for starting communication, addressing the Target and sending the commands and data. The Target is responsible for controlling the phase lines for data

transfer, and for ending the communication session. The communicators on the bus can be Initiators Only, Targets Only, or both Initiators and Targets. Every communicator on the SCSI bus is given an address from 0 to 7, and each address has an associated priority with 0 the highest priority. The address is used by Initiators for selection of Targets, and the address priority is used for multiple Initiators attempting concurrent bus access

The SCSI bus operates with 8 phases of operation during a communication session, all of which need not be used. The 8 phase are: BUS FREE, ARBITRATION, SELECTION, RESELECTION, COMMAND, DATA, MESSAGE, and STATUS [1]. BUS FREE indicates that no activity exists on the bus and that an Initiator can start communication. ARBITRATION is used by multiple Initiators to determine priority. SELECTION is the phase whereby an Initiator uses the SCSI bus to select a Target communicator. RESELECTION supports Initiators who break communication with the Target on a time consuming processes to free the bus. A Target will use RESELECTION to inform an Initiator that that the started process is complete. The COMMAND, DATA, MESSAGE, and STATUS phases are used for data transfer.

A typical bus access begins with a BUS FREE phase [1]. An Initiator begins an ARBITRATION phase, allowing other Initiators the opportunity to vie for the SCSI bus. Once ARBITRATION is complete, the winning Initiator begins the SELECTION phase, placing a Target address on the

SCSI bus. The Target acknowledges its selection, and communication continues. The Target now has control of the SCSI bus, setting phase lines for the COMMAND, DATA, and STATUS phases, and beginning any handshake transfers. Once the Target obtains or sends the data, the BUS FREE phase is entered once more. For the Priority Event Monitor, the RESELECTION and MESSAGE phases are not used and were not included in the above example.

3.4.4 Logical SCSI Interface Simplification

The SCSI bus phases of operation are simplified by Circia [3] to be simplified to BUS FREE, ARBITRATION, SELECTION, and Information Transfer Phases. An alternate simplification of the SCSI transfer phases is proposed for the purposes of data transfer rate analysis. This simplification forms the SCSI bus phases into ARBITRATION/SELECTION phase, COMMAND phase, and DATA phase. This breakdown makes sense when trying to account for overhead vs. data rate transfer time. The BUS FREE phase is removed because it does not directly affect timing of data transfers, other than those times included in the ARBITRATION/SELECTION phase. The ARBITRATION and SELECTION phases are joined because they represent system overhead for establishing communication. The COMMAND phase is considered

communication overhead for data transfer and the DATA phase accounts for data transfer time to or from the Initiator. The MESSAGE phase is not used in the existing design for the Priority Event Monitor, and the STATUS phase is bundled with the COMMAND phase timing as overhead. This simplification allows for an easy calculation of system and communication overhead verses data transfer time.

3.5 Personal Computer Control

The link between the PS/2 and the Monitor has now been defined as the SCSI bus and the SCSI bus's logical interface is described in the previous section. The control and communication between the PS/2 and the Monitor will now be described.

Both the PS/2 and the Monitor operate as SCSI Initiators and Targets, allowing the Monitor to initiate communication with the PS/2. If the Monitor was only capable of being a Target then data transfer would always have to be initiated by the PS/2. This could cause data loss as the Event Rate increased. By allowing both the Monitor and the PS/2 to be Initiators and Targets on the SCSI bus, the problem of data transfer control is passed to the Monitor. The PS/2 is Initiator for sending commands and retrieving status and the Monitor is Initiator for sending data to the PS/2.

Six commands are implemented to allow the PS/2 to control the Monitor. When issuing these commands, the PS/2 acts as the SCSI Initiator and the Monitor acts as Target. The six commands are: Set_Control, Reset_Monitor, Start_Recording, Stop_Recording, Send_Status, and Ack_Connection. The Set_Control command is used to set the parameters concerning which Events the Monitor is to capture. Its parameters determine whether the Priority Level Changes, Wait Events, or both are captured. The Set_Control command is the only command available for setting the capture control of the Monitor.

The Reset_Monitor Command is used to set all parameters to default values, clear memory, and stop any Event recording currently taking place. The Start_Recording command is used to begin a capture session using the current capture settings. The Stop_Recording command is used to stop capturing Events, leaving the current capture settings in place. The Reset_Monitor, Start_Recording, and Stop_Recording commands are used for control of Monitor operations. The Send_Status command is used to retrieve status information from the Monitor concerning current capture settings, memory problems, etc. The Ack_Connection command is used to assure communication with the Monitor. Neither the Send_Status and the Ack_Connection command affect control settings on the Monitor and are only used to obtain needed information.

With these six commands, control of Monitor operations is achieved. Now discussion turns to Monitor control of the PS/2. Two commands are implemented to allow the Monitor to control the PS/2. When issuing these commands, the Monitor is the SCSI Initiator and the PS/2 is the SCSI Target. The two commands are `Receive_Data` and `Ack_Connection`. The `Receive_Data` command is used to transfer Event Records from the Monitor to the PS/2. This command is used to fulfill the Monitor function. The data for this command is a packet containing Event Records captured by the Monitor using the settings from the PS/2 command `Set_Control`. The `Ack_Connection` command has the same function for the Monitor as for the PS/2. It is used simply as a way to establish that communication with the PS/2 exists and is available.

This is the high level view of communication and control between the PS/2 and the Monitor. The commands described for the PS/2 and the Monitor fulfill the needs for information transfer between both systems, allowing the PS/2 to control Monitor operations, and allow the Monitor to transfer Event Record data back to the PS/2 where they can be stored into a file. The functions of the Monitor and the PS/2 are satisfied. The software to implement the host control of the Monitor is written in the C programming language, and a copy of the software is given in Appendix C.

CHAPTER 4

MONITOR HARDWARE ARCHITECTURE

The Priority Event Monitor, as shown in Figure 3.1, is divided into two major functional groups. A PS/2 performs the functions of host and controller while an external Monitor performs the functions of event capture, timestamp, and data transfer. The architecture and design of the Monitor is discussed in this chapter. Discussion begins with the partitioning of the Monitor followed by a discussion of the design of each partition.

4.1 Monitor Architecture

The Monitor is responsible for three major tasks in the Priority Event Monitor, recognizing and capturing Events, timestamping Events, and transferring Event Records to the PS/2. To perform these functions, the monitor has an interface to the 4702 computer for Event recognition and capture and a SCSI bus interface to the PS/2 Event Record transfer and Monitor control. The Monitor is implemented as external hardware with a representative picture shown in Figure 3.2.

To perform its tasks, the Monitor is partitioned into seven modules. Figure 4.1 gives an overview of the monitor with each module shown.

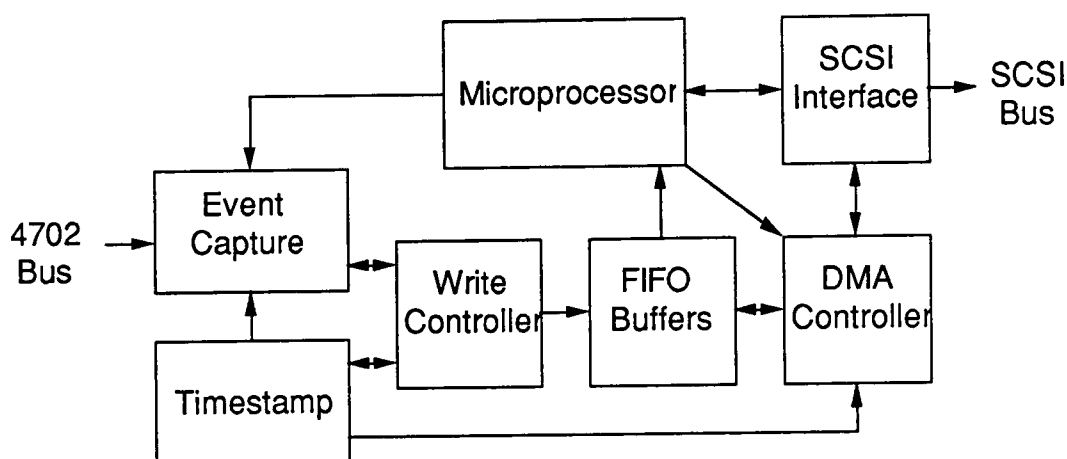


Figure 4.1: Monitor Architecture. Signal directions represent communication and control between modules.

The SCSI Interface implements the physical interface for communication between the Host and the Monitor. The Microprocessor is responsible for the logical communication interface with the PS/2 and for global control of Event capture. The Microprocessor is also responsible for initiating the Event Record transfers to the Host. The FIFO buffer is used to store Event Records until an acceptable number of Event Records is stored and the SCSI bus is available. The DMA Controller is used to transfer Event Records from the FIFOs to the SCSI bus interface, bypassing the microprocessor. These four modules, SCSI Interface, Microprocessor, FIFO Buffer, and DMA Controller, form the communication and control modules for the Monitor. The Event Capture module is responsible for observing and recognizing Events on the 4702

busses. The Timestamp module controls timestamping Events and overall timing and clock generation needs. This module is also responsible for generation of Timer Overflows. The last module discussed, the Write Controller, is used to form and write Event Records from the Event Capture and Timestamp modules into the FIFO buffer.

These seven modules implement all functional needs for the Monitor and complete the hardware needs of the Priority Event Monitor. Each of the seven modules will be discussed in greater detail in the following sections.

4.2 FIFO Buffers

Buffering on the Monitor is needed for two reasons. First, to act as a data rate translator and second to store Event Records for data packet formation and SCSI availability. Events on the 4702 occur at irregular times and are not predictable, whereas a SCSI transfer should be organized for higher use of the SCSI bus bandwidth. Also, the SCSI bus is an arbitration based, multi-user bus so access time may vary. For these reasons data buffering is needed.

The FIFO buffer has interaction with three modules of the Monitor. It acts as an intermediary between the Write Controller and the DMA Controller, and also interacts with the Microprocessor. The Write

Controller is responsible for writing Event Records to the FIFO and the DMA Controller is responsible for reading the buffer when a SCSI data transfer is occurring. The FIFO buffer interfaces to the Microprocessor to indicate the proper amount of accumulated data for a SCSI transfer. The block diagram of the FIFO buffer is shown in Figure 4.2 below.

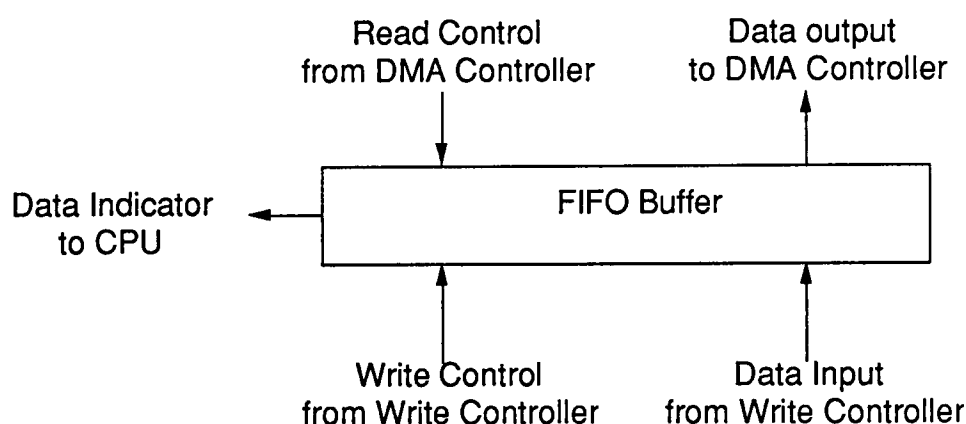


Figure 4.2: Block Diagram for the FIFO Buffer. Interactions with the DMA Controller, Write Controller and CPU are shown.

Notice in the Figure that the structure of the FIFO buffer, which is determined by its width and depth, is not indicated. The remainder of this section is devoted to determining the structure of the FIFO buffer and to the description of the parts used to form the buffer.

The minimum length of the SCSI data packet is set in Section 4.7.2 to be 512 bytes per packet, setting the depth-width product for the FIFO buffer. To determine the values of depth and width, the record rate, Event Record length, and Write Controller complexity will be considered. The

Write Controller is responsible for writing data from the Event Capture and Timestamp modules into the FIFO buffers. With a Record Rate of 2 Events per microsecond for an Event Record length of 4 bytes, the Write Controller will be responsible for writing a maximum of 8 bytes in 1 microsecond. This rate affects the write speed needed for the FIFO buffers and the control speed needed for the Write Controller. If a byte wide FIFO is used, the Write Controller will be responsible for writing, and the FIFO will be responsible for accepting, 1 byte every 125 nanoseconds. If the width of the FIFO is increased, the write speed for the Write Controller and the FIFO buffers will be decreased, yielding an easier and more stable design. Each write of an Event Record to the FIFO takes several steps. For the Write Controller to accomplish the steps outlined above using a byte wide FIFO buffer is impractical (the Write Controller is covered in detail in Section 4.6). The control needed would require a high Write Controller clock frequency. To reduce the clock speed and control needed, the FIFO width is increased to accommodate a more reasonable Write Controller clock speed. By increasing the width of the FIFO buffer to 4 bytes, an entire Event Record can be written at once, requiring the Write Controller to only write twice each microsecond.

With the width of the FIFOs set at 4 bytes, the depth can be determined from the SCSI data packet size. The data packet size for the SCSI bus was determined to be 512 bytes in length (the SCSI bus packet

size is discussed in Section 4.7.2). This means the FIFO must store 128 Event Records before a complete SCSI data packet is ready. To allow for Event Record accumulation during SCSI transfer of data, the FIFO buffer depth will be doubled to 256 bytes, with the data packet sent when 128 bytes are accumulated. Most FIFOs have a flag indicating half full and this signal will be used to indicate to the Microprocessor that a DMA transfer is needed.

The structure for the FIFO buffer is now complete. The buffer is 4 bytes wide and 256 bytes deep. The write control of the buffer is controlled by the Write Controller and the read control is handled by the DMA Controller. DMA transfer is then controlled by the CPU and the need for data transfer is indicated with a line indicating that the FIFO is half full.

The FIFO buffer is formed using four Am7200-80 FIFOs from Advanced Micro Devices [5]. These devices have read and write access times of 80 nanoseconds, well within the needed access time for the Write Controller, are 256 bytes deep, and have several status flags, one of which is the Half-Full flag needed to indicate a need for DMA transfer to the CPU.

FIFO memory parts lend themselves easily to increased width, so the 4 byte by 256 byte FIFO buffer needed for the monitor is easily realizable. The Write Controller must simply interface to one signal, /W, and the DMA Controller must interface to the /R signal, for writing and reading respectively.

4.3 Event Capture

The Event Capture module is responsible for observing the 4702 busses and logging all Events. When Events occur the Event Capture module sets flags for the Write Controller. The 4702 bus signals and the signatures that indicate Events are described in detail in Section 2.2 and 2.3 respectively. This section covers the design of the Event Capture module.

The Event Capture module has three main interactions. The first interaction is with the 4702 bus for Event observation, the second is with the Write Controller for Event Indication and the third is with the Microprocessor for overall control of Event capture. A block diagram for the Event Capture module is given in Figure 4.3.

The Event Capture module observes the 4702 busses and sets indicators for each Event: the Wait Entry indicator (WEN), the Wait Exit indicator (WEX), and the Level Change indicator (LC). The Write Controller is responsible for resetting the indicator after the respective

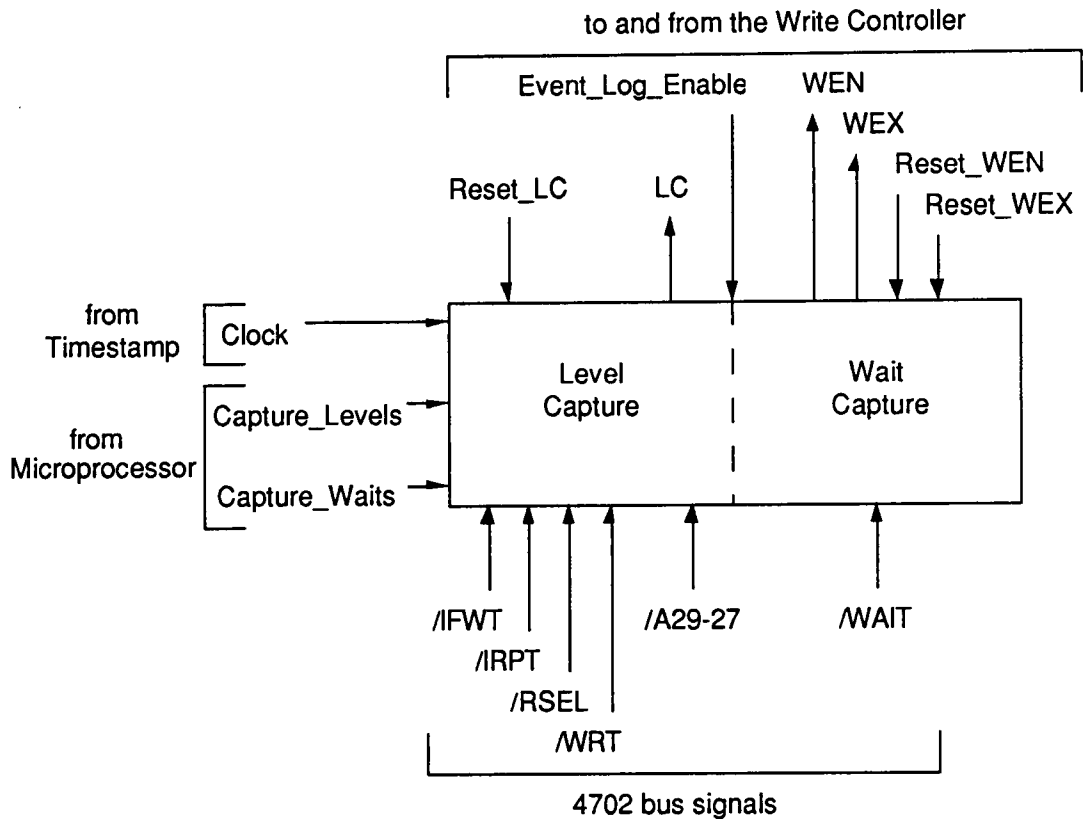


Figure 4.3: Event Capture Module Block Diagram. Connections with the Write Controller, Timestamp and Microprocessor modules and with the 4702 address and control bus are shown.

Event Record write. Therefore three Event reset signals, Reset_LC, Reset_WEN and Reset_WEX, are sent from the Write Controller to the Event Capture module. The Write Controller is also responsible for freezing action on the Event Capture modules to mark the beginning of each microsecond. This is accomplished using the Event_Log_Enable signal. The Microprocessor controls two signals, Capture_Levels and Capture_Waits, allowing individual capture control of Level and Wait Events.

Note that some signal preprocessing has been designed by IBM for use with the 4702 bus. This circuitry is called the Redrive Card and is used to increase the driving capability of the SOLO microprocessor bus. The Redrive Card contains initial decoding hardware for Wait Events, and is presented to the Event Capture module as a signal called WAIT. The WAIT signal is active when the 4702 is in the Wait state, and is inactive when the 4702 is not in Wait state.

The Event Capture module is further divided between Wait Event Captures and Level Event Captures which are each described below. The design of the Wait Event Capture will be discussed first followed by the design of the Level Event Capture.

4.3.1 Wait Event Capture

The Wait Event Capture is handled by a small state machine clocked at 16 MHz. The state machine implementation is effective at observing the 4702 bus and the clock speed allows adequate sampling. The block diagram of the Wait Capture module is shown in Figure 4.4 below.

The Wait machine has as its input the WAIT signal from the 4702 Redrive Card, the Enable_Waits signal from the Microprocessor, and the Reset_WEN and Reset_WEX signals from the Write Controller. The Wait machine outputs the WEN and WEX signals indicating that a Wait Entry or

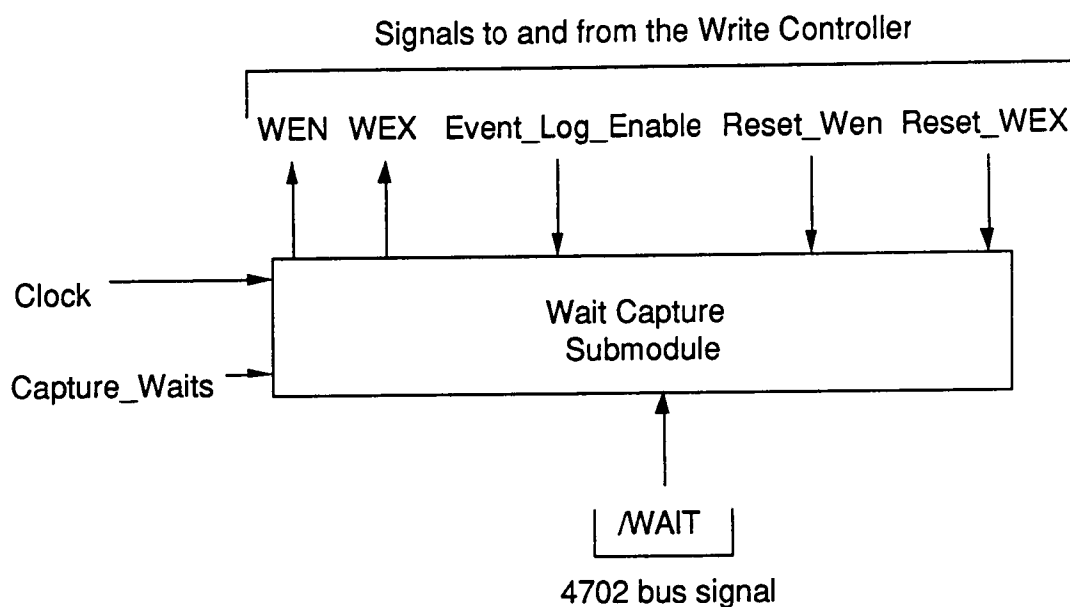


Figure 4.4: Wait Capture Block Diagram. Write Controller and 4702 bus signals are highlighted. The Clock and Capture_Waits signals are from the Timestamp and Microprocessor modules respectively.

Wait Exit Event has occurred. When the WAIT signal makes a transition from inactive to active, the Wait machine activates the WEN line. The Write Controller, after the WEN Event Record has been written into the FIFO buffers, will activate the Reset_WEN signal, and the Wait machine deactivates WEN. When the WAIT signal makes a transition from active to inactive, the Wait machine activates the WEX line to indicate that the 4702 has exited the wait state and is now actively processing once again. When the Write Controller has written the Wait Exit Event Record into the FIFO buffer, the Reset_WEX line is activated and the the Wait machine deactivates WEX. The Wait machine now watches for the next active Wait Entry. The state diagram for the Wait machine is given in Figure 4.5.

The machine given in Figure 4.5 implements the needed functionality for Wait capture. State S1 is the start state for the machine. When the WAIT signal activates, the machine makes a transition to S2 and activates WEN. If the Reset_WEN signal is received, the machine goes to S3, and awaits the deactivation of WAIT. If in state S2, the WAIT signal deactivates before the Reset_WEN, the machine makes a transition to S4 and activates the WEX signals as well as the WEN signal. From State S3, the machine waits for a deactivation of WAIT, then makes a transition to S5 and activates the WEX signal. From State S4, the machine watches for an active Reset_WEN, then transitions to S5. From S5, the Wait capture machine watches for an active Reset_WEX, then makes a transition to S1, and again waits for the WAIT signal to activate.

Wait capture is accomplished through the use of one state machine. The state diagram given in Figure 4.5 accomplishes the goals of Wait Capture, implementing the recognition of both Wait Exits and Wait Entries. The PALASM description file for the state diagram given in Figure 4.5 is in Appendix D.

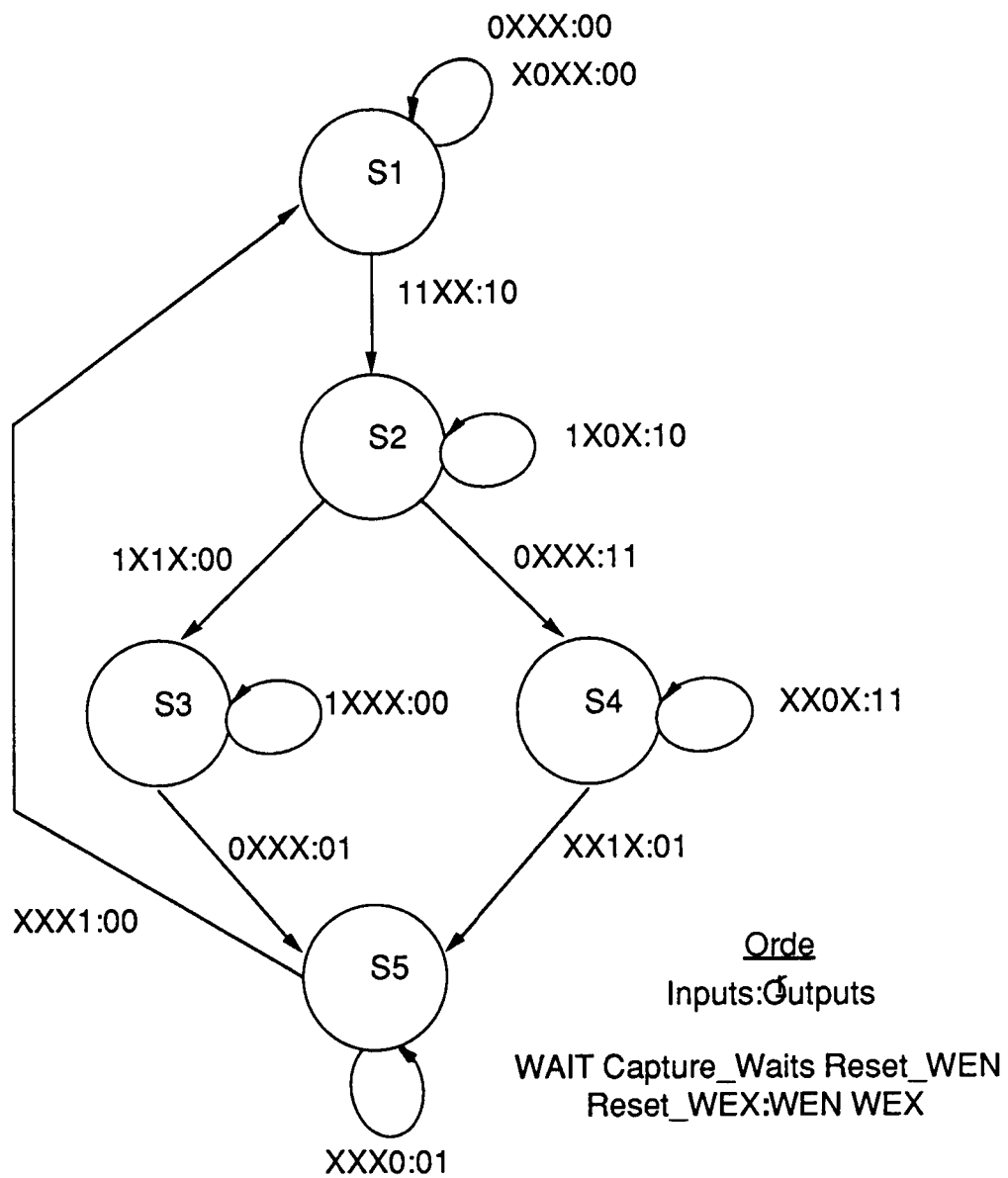


Figure 4.5: Wait Machine State Diagram.

4.3.2 Level Event Capture

This section describes the partition and design of the Level Capture module. As described in Section 2.3, capturing the Priority Level Change consists of capturing two values, LPL and CPL, from the 4702 address bus. These two values must be captured to record the Priority Level Change Event. The block diagram for Level capture is shown in Figure 4.6.

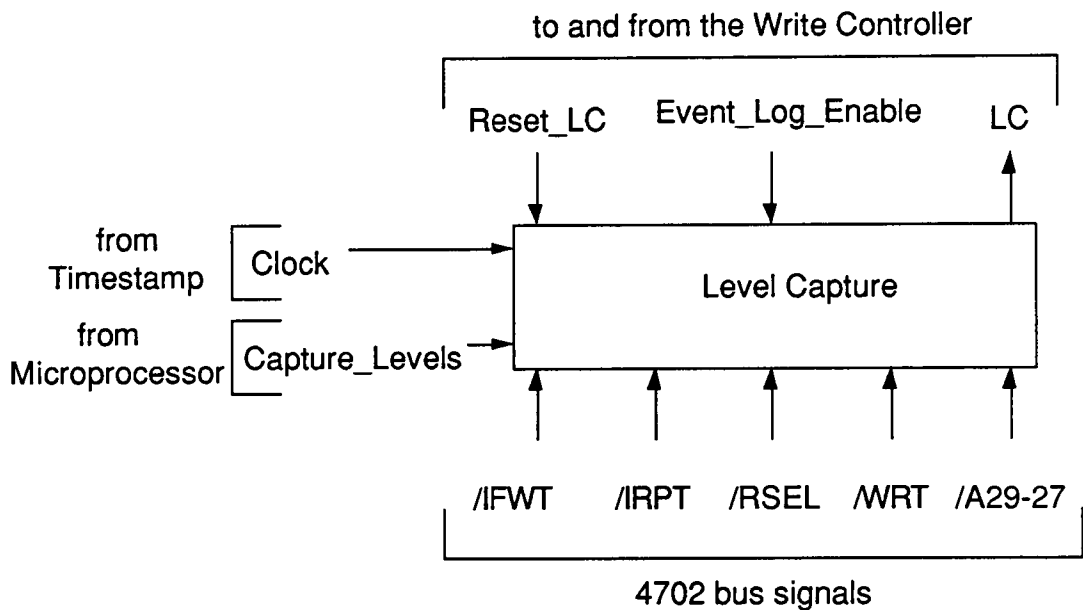


Figure 4.6: Block Diagram for Level Capture. Signals from the Write Controller, Timestamp and Microprocessor modules and the 4702 bus signals are shown.

The Level capture has as its input four 4702 control signals, /IFWT, /IRPT, /WRT, and /RSEL, and three address bus signals, /A29, /A28 and /A27. The 4702 control signals are used to determine when the LPL and CPL values are available on the 4702 address bus. The Level capture

accepts the Capture_Levels signal from the microprocessor and the clock signal from the Timestamp module. The Level capture outputs LC to and accepts Reset_LC and Event_Log_Enable from the Write Controller. Level Capture consists of several small state machines clocked at 16 MHz. The LPL and CPL values are available on the 4702 address bus for approximately 290 nanoseconds each, meaning that the 16 MHz clock is adequate.

The Level capture is accomplished by first capturing the LPL value, then capturing the CPL value. Once the CPL value is captured, the Level Change is logged and the Level capture activates LC. The LC signal and the LPL and CPL values are available to the Write Controller. When the Write Controller writes the Priority Level Change, it activates Reset_LC and the Level capture deactivates LC and waits for the next Event. The partition of the Level Capture is shown in Figure 4.7.

The LPL and CPL decoder determines when the LPL and CPL values are available on the 4702 bus (as determined by the Event signatures given in Section 2.3). The LPL and CPL machines capture the LPL and CPL values into their respective registers and the LC machine is used to communicate with the Write Controller, indicating a Priority Level Change Event. The Priority Level Change sequence is ended by the capture of the CPL value, with the CPL_Available signal used to trigger the LC machine. The state diagrams of the LPL, CPL, LC machines are now be discussed.

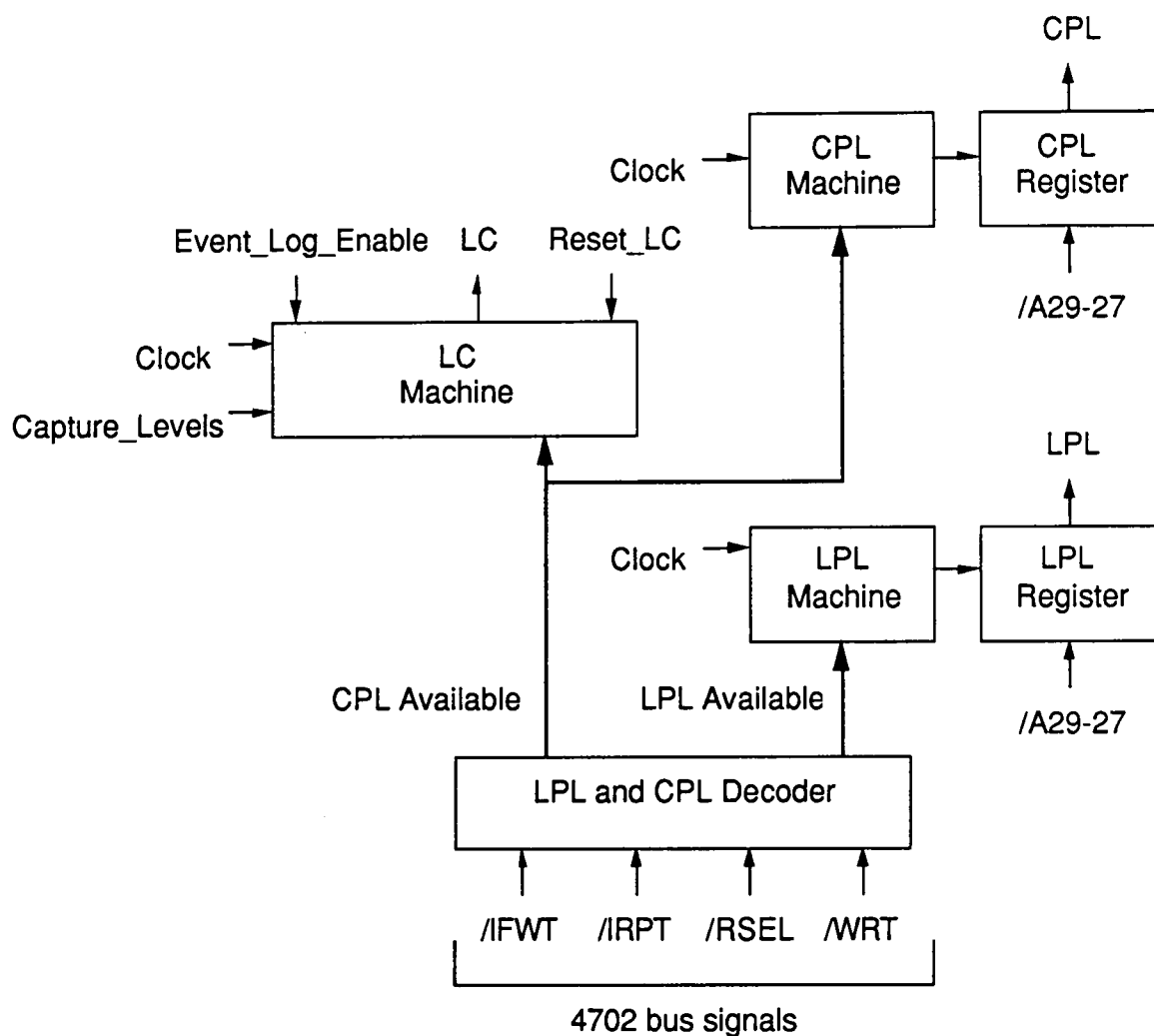


Figure 4.7: Event Capture Module - Level Capture. This diagram shows the functional blocks needed to capture Priority Level Changes from the 4702 address and control busses.

The LPL machine has as its input the signal LPL_Available, and outputs a signal LPL_Clock which clocks the three bit LPL register whenever the Last Priority Level information is on the 4702 address bus. The state diagram for the LPL machine is given in Figure 4.8.

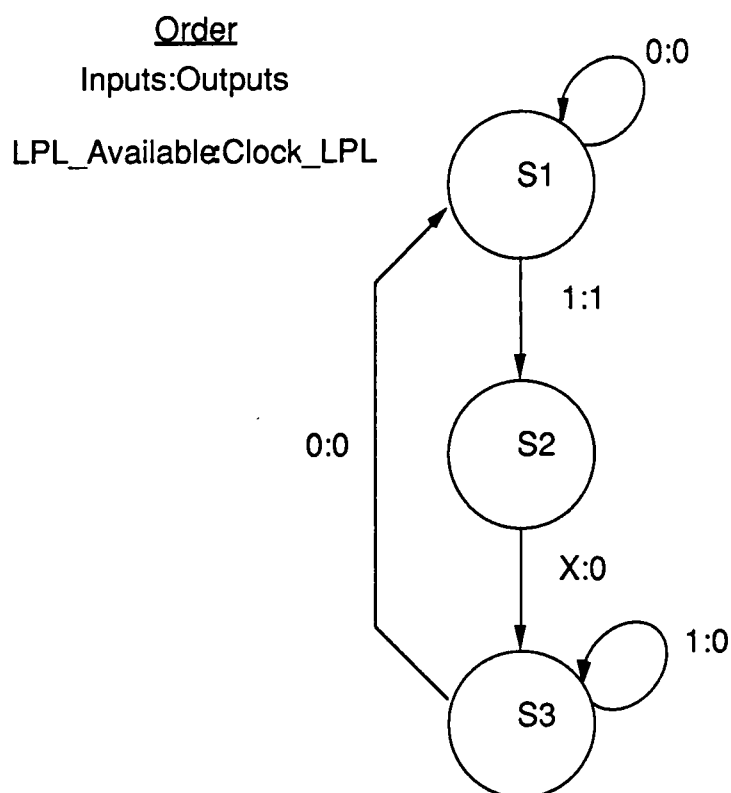


Figure 4.8: LPL State Diagram. The use of the LPL and CPL Decoder greatly simplifies the state machine requirements needed to capture the LPL value.

The LPL machine remains in the initial state, S1, until the signal LPL_Available is active. At this point the machine makes a transition to state S2 and activates Clock_LPL. The LPL value is now stored into the LPL register for use by the Write Controller. The LPL machine now makes a transition to state S3 deactivating Clock_LPL. The machine remains in state S3 until the LPL_Available signal is no longer active. At this point the machine transitions back to the start state of S1.

The CPL machine state diagram is given in Figure 4.9. Notice the machine is the same as the LPL machine and its explanation is similar.

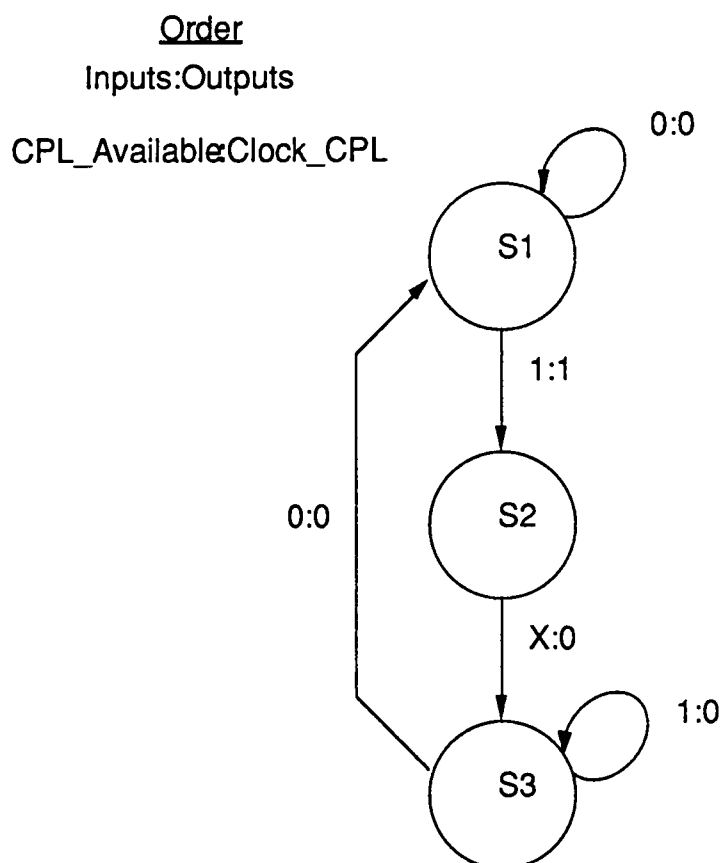


Figure 4.9: CPL Machine State Diagram. The use of the LPL and CPL Decoder greatly simplifies the state machine requirements needed to capture the LPL value.

The only differences are the input and output signals. For a full discussion of the machine characteristics, see the LPL state machine explanation given above.

The LC machine has as its inputs CPL_Available from the decoder and Reset_LC from the Write Controller and has an output of LC indicating that a Priority Level Change has occurred. A master control signal from the Microprocessor is also input to the LC machine. This signal, Capture_LC, must be active to log any Priority Level Changes.

During a Priority Level Change, the CPL is the last to be available on the 4702 bus, therefore it represents the end of the Priority Level Change. Once the CPL value has been captured, the LC machine activates the LC signal, informing the Write Controller. The Write Controller writes the Event Record and activates Reset_LC, causing the LC machine to deactivate the LC signal. The LC machine will then wait for the next Priority Level Change to occur. The state diagram for the LC machine is given in Figure 4.10.

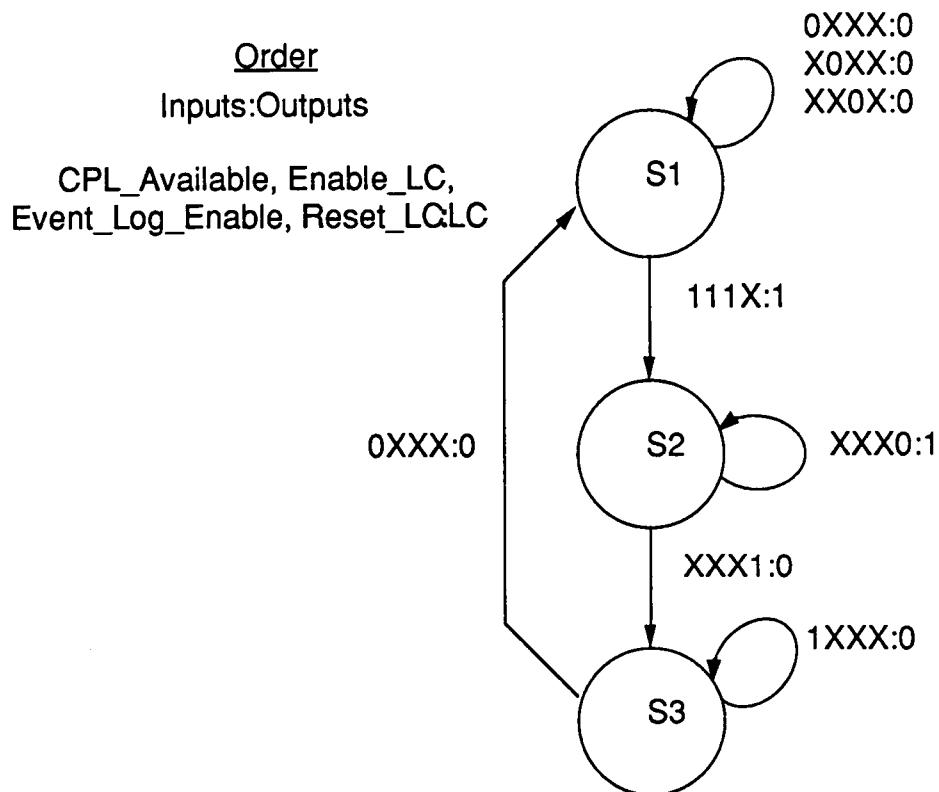


Figure 4.10: LC State Diagram. The use of the LPL and CPL Decoder greatly simplifies the state machine requirements. Notice that the LC machine control is greater than that needed for the LPL and CPL machines

State S1 represents the initial state. The LC machine makes a transitions to state S2 only when the Enable_LC signal and the CPL_Available signal are both true, and during this transition, will activate the LC signal. Transition from state S2 to S3 occurs when the Write Controller activates the Reset_LC signal, and the LC signal is deactivated. The machine stays in state S3 until the the CPL_Available signal is deactivated by the CPL and LPL decoder, ending the session. At this point, the machine makes a transitions to state S1 and awaits the next Priority Level Change.

The LPL and CPL registers are three bits each, capable of containing the LPL or CPL values from the 4702 Address bus. The registers are clocked by the LPL and CPL machines when their respective values are available. The LPL and CPL decoder equations are given in Section 2.3. The PALASM files for the LPL, CPL, and LC machines are given in Appendix D. This completes the design of the Event Capture module.

4.4 Timestamp Module and Timer Overflow Detection

This section describes the Timestamp module as well as the Timer Overflow Event detection hardware. The Timestamp module interfaces to the Write Controller by providing the Timestamp value for Event Records

and by providing the Timer Overflow Event indicator. The Timestamp module also provides the clock signals to the Event Capture, the Write Controller, and the DMA Controller. An independent clock is used for the Microprocessor allowing for clock isolation and test purposes. The Timestamp Module has one input signal, the Reset_TO signal from the Write Controller used to reset the Timer Overflow (TO) signal. The block diagram for the Timestamp module is given in Figure 4.11. The Module consists of four main modules: the clock generator, the clock divider, the Timer stamp counter, and the Timer Overflow detection circuit.

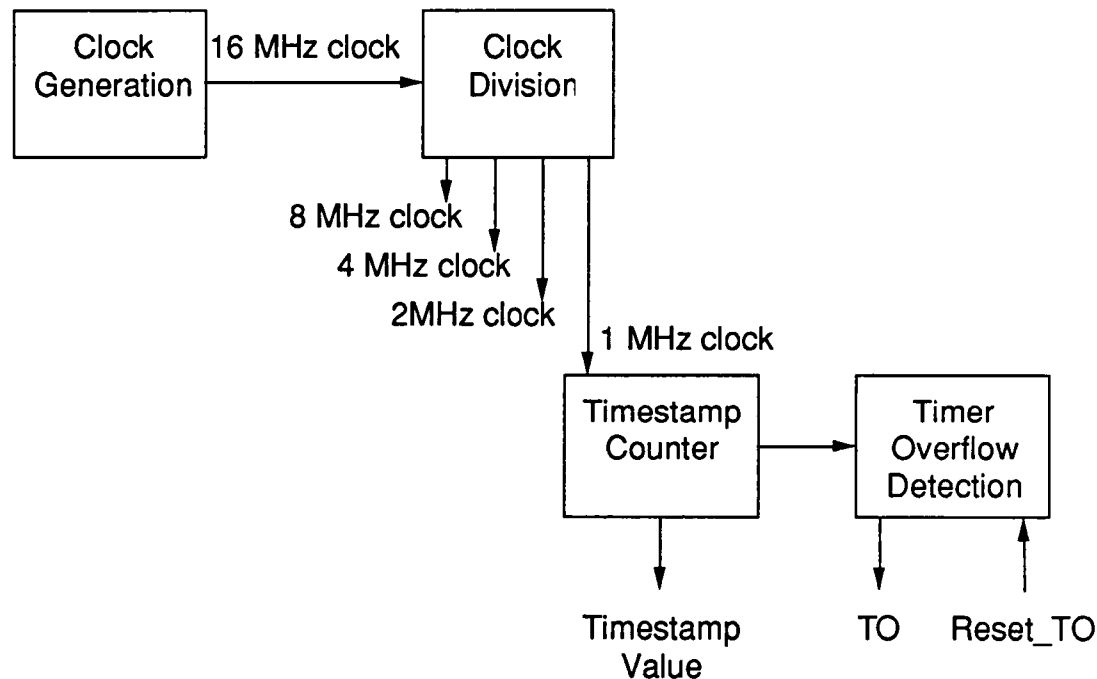


Figure 4.11: Block Diagram of the Timestamp Module.

The fastest clock speed for the system is set in Section 4.4 to be 16 MHz, needed for the Event Capture state machines (and for the Write Controller). This clock frequency easily yields a 1 MHz clock through simple clock division and the 1 MHz clock is used to drive the Timestamp counter. The values of 8, 4, and 2 MHz are also derived from the 16 MHz clock for possible use (The 8 MHz signal is used in the DMA Controller). The Clock Generator shown in Figure 4.10 consists of a 16 MHz oscillator to generate the initial clock signal. Clock division is performed by a fast 4 bit binary counter, with bit 0 forming the 8 MHz signal, bit 1 the 4 MHz signal, etc. Bit 4 of the Clock divisor is used to drive the Timestamp Counter.

The length of the Timestamp is set at 16 bits in Appendix B. The 16 bit counter needs to be fast to allow for quick settling of the output value so the Write Controller can have fast access to the count data. The Timestamp counter and the Clock divider are both implemented using 74LS163 4 bit counters, with the Timestamp counter utilizing a built-in look ahead carry mode. With this implementation, the delay through the Timestamp counter is 29 seconds. The delay for the clock divider is also 29 nanoseconds, yielding very little delay between the 16 MHz clock and its clock derivatives. When the 16 MHz clock is used in conjunction with its derivatives, the delay has been accounted for.

The Timer Overflow detection circuit monitors the Timestamp Counter to detect changes from hex FFFF to 0 indicating a rollover or an overflow, setting the TO signal (for Timer Overflow). When the Write Controller writes the Timer Overflow Record, Reset_TO is activated, forcing reset of the TO signal. The Timer Overflow detection circuit is formed using a D flip flop. The D input is tied high and the flip flop clock is tied to the ripple carry output of the last stage of the Timestamp Counter, and the reset signal tied to the Reset_TO signal.

This completes the explanation of the Timestamp module. The Clock Generator is a 16 MHz oscillator, the Clock Divider is a fast 4 bit counter, the Timestamp Counter is a fast 16 bit counter, and the Timer Overflow circuit is a D flip flop. The schematic for the Timestamp module is available in Appendix E.

4.5 Write Controller

The Write Controller has two tasks, recognizing Event flags from the Event Capture and Timestamp modules and writing Event Records into the FIFO buffer. The Write Controller interfaces to the Event Capture module for Priority Change and Wait State Indicators and interfaces to the Timestamp module for clock and for the Timer Overflow Indicator. The Write Controller interfaces to the FIFO buffers for storing Event Records.

The Write Controller is responsible for writing multiple Event Records into the FIFO memory for each microsecond, specified as the Record Rate in Section 2.4. A Write Controller cycle begins with the Write Controller freezing changes to the Event Indicators in the Event Capture and Timestamp modules. The Write Controller then checks to see if any Event Indicators are active. If not, then the Event Indicators are unfrozen. If an Event Indicator is active, the Write Controller writes the Event Record into the FIFO buffer, then resets the Event Indicator. After resetting the Event Indicator, the Write Controller checks for another active Event Indicator, and the process begins again.

4.5.1 Write Controller Overview

The Write Controller block diagram is shown in Figure 4.12. The Write Controller has four Event Indicator inputs, Timer Overflow (shown as TO in Figure 4.12), Priority Level Change (LC), Wait Entry (WEN), and Wait Exit (WEX), and two clock signal inputs, the 16 MHz clock and the WR_EN signal. Additional inputs include the LPL and CPL values from the Event Logging module, needed for forming the Comment byte for the Priority Level Change Event Record.

The Write Controller has four Event Indicator reset outputs associated with each of the Event Indicator inputs and an output signal, Event_Log_Enable, used to freeze Event Indicator action. The Write Controller output FIFO_Write is used to write Data to the FIFO buffer. In addition the Write Controller outputs four byte Event Records to the FIFO buffers. The Timestamp bytes are simply passed directly from the Timestamp module and the Command and Comment bytes are converted from the Event Indicators.

The Write Controller is composed of six blocks: the Encoder, Event register and Decoder, a state machine for control, and two converters for the Command and Comment Bytes. The Event Indicators, TO, WEN, WEX, and LC are inputs to the Priority Encoder, which allows a reduction in the number of lines needed to represent an Event. The encoder also feeds a signal, Event_Occurred, to the state machine, to indicate an active Event Indicator.

The Event register is clocked once for each Event Record written, as the encoder output changes with Event Indicator reset. The Decoder uses the Event register output to reset the appropriate Event Indicator. The State Machine also controls the FIFO_Write signal for writing data to the FIFO memory. The Command and Comment Converters simply take the Event

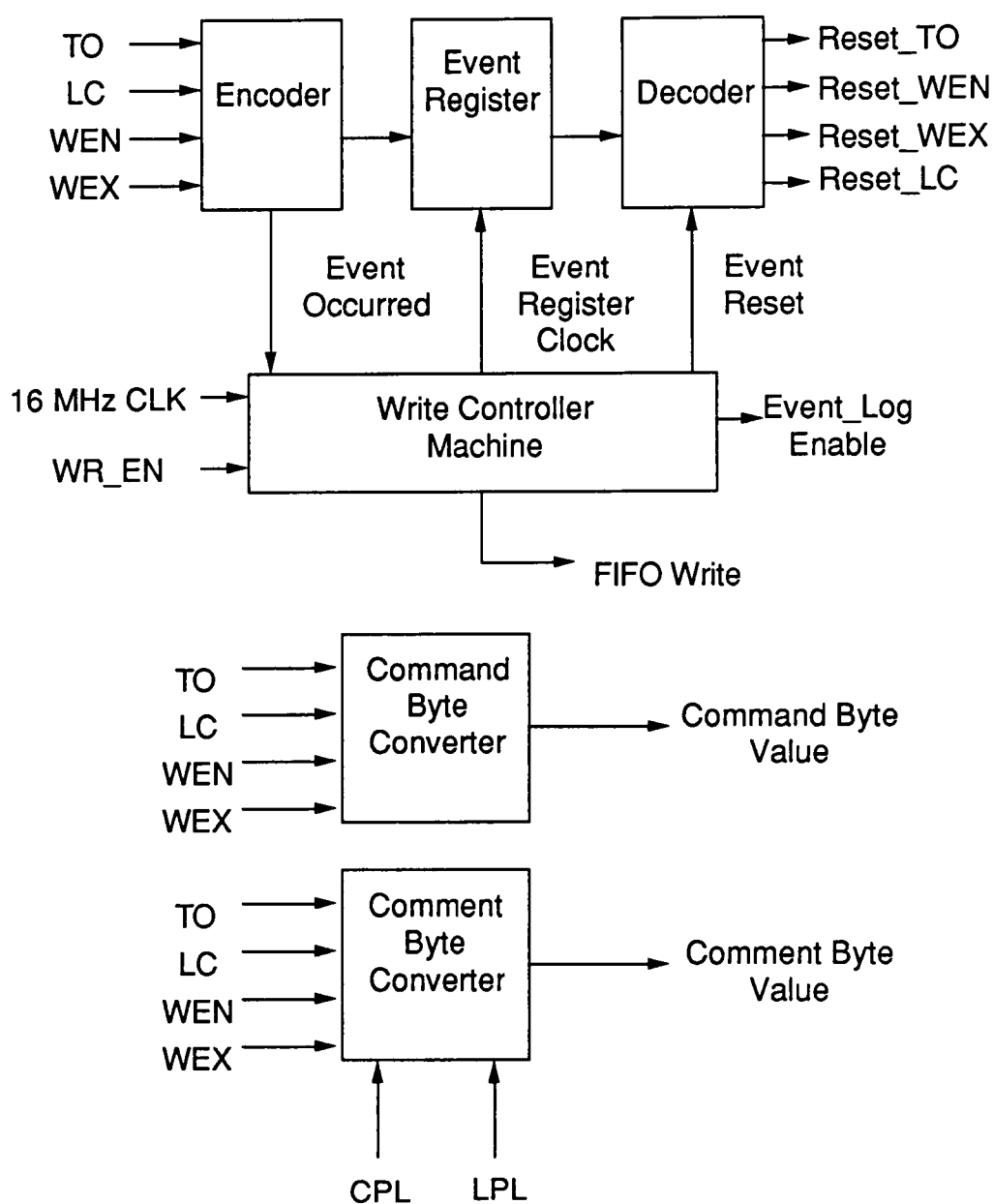


Figure 4.12: Write Controller Block Diagram.

Indicators as input and then outputs the correct byte values for the Command and Comment Bytes. The six blocks, the Command and Comment Converters, the State Machine, the Encoder, the Event Register and the Decoder form the Write Controller.

4.5.2 Write Controller Implementation

To describe the Write Controller State Machine in terms of its state diagram, the implementation of all Write Controller blocks must be discussed, as the state diagram is timing dependent. Covered first will be the implementation of the encoder, register, and decoder for recognizing and resetting Events. Following this will be a discussion of the state machine implementation and the implementation of the Command and Comment converters. A discussion of the timing of the FIFO memories is given in Section 4.2 and a timing discussion for the Timestamp values is given in Section 4.4.

The Write Controller is responsible for writing two Event Records every microsecond, specified by the Record Rate given in Section 2.4. The FIFO buffer is constructed so that the speed needed for the Write Controller is reduced but the Write controller is still responsible for several operations for each written Event Record. To reduce the delays, FAST logic is used in the implementation of the Encoder, Register

and Decoder. The Encoder is implemented as a 74F148 Priority Encoder, the Event Register as a 74F175, and the Decoder as a 74F138 Decoder.

The signals, TO, LC, WEN, and WEX are Encoder inputs and the Encoder outputs are inputs to the Event Register. The Encoder also outputs the Event_Occurred signal for the State Machine. The Event Register outputs are used as the inputs to the decoder, and the Event Register is clocked by the Clock_Event signal from the state machine. The Decoder outputs are connected to Reset_TO, Reset_LC, Reset_WEN, and Reset_WEX to reset the Event Indicators after writing of the Event Record. An enable signal is connected to the Reset signal from the State Machine allowing the State Machine to control Event Indicator reset. The implementation of the state machine of the Write Controller is accomplished using a PAL16R8 and the Command and Comment Converters are PAL16L8s. The implementation of the Write Controller is shown in Appendix E.

The connections and timing of all parts of the Write Controller have been described. Following is a description of the major timing delays for the Write Controller. The major timing delays are discussed and a value for each delay is given. Once these timing values are known, the State Machine for the Write Controller can be designed and the description of the Write Controller will be complete.

4.5.3 Write Controller State Machine

The state diagram for the Write Controller state machine is directly dependent on the timing constraints introduced by the Write Controller implementation. In the following paragraphs, the timing constraints for the Write Controller will be defined and times given to each constraint. The constraint times will then be converted to state times to facilitate state machine design (a state time for the 16 MHz clock is 62.5 nanoseconds). Following the constraint definitions, the state diagram will be given and explained.

There are three major areas of timing to consider for the operation of the Write Controller: FIFO write setup times, Event Indicator reset times, and clock pulse width times. The FIFO setup times will be discussed first as they pertain to writing the first Event Records. The Event Indicator reset timing will then be discussed as it pertains to timing for the second or more Event Indictors. Following this is a brief discussion of the two pulse width times.

An Event Record, the Command and Comment bytes and the 16 bit Timestamp value, must abide with the 40 nanoseconds setup time for the Am7200-80 FIFO buffers. The Command and Comment bytes follow one data path and the Timestamp follows another, therefore the setup time analysis must check both paths. The delay values for the Command and

Comment bytes is defined as T_{ccb} and the delay for the Timestamp is defined as T_{tmp} . T_{ccb} is a function of the stability of the Event Indicators and the propagation delay of the Command and Comment byte converters. The Event Indicator's stability depends on `Event_Log_Enable` which in turn depend on the `WR_EN` signal. Figure 4.13 shows the timing diagram for the `WR_EN` signal interaction with the 16 MHz clock, the `Event_Log_Enable` signal, and the Event Indicators.

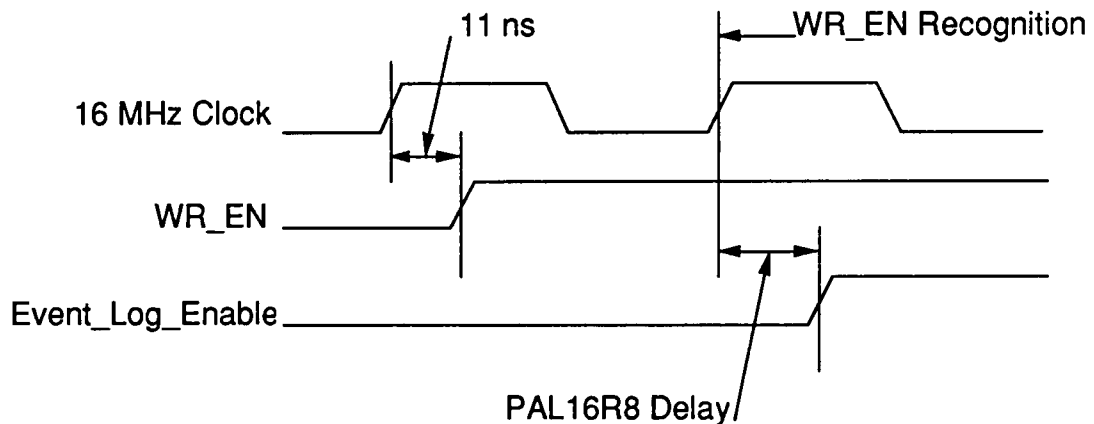


Figure 4.13: Timing of the `WR_EN` signal.

T_{ccb} therefore depends on a PAL16R8 register delay (for the Event Logging modules) and the delay through the Command and Comment Byte Converters, or the full delay through a PAL16L8. These delays are 15 nanoseconds and 25 nanoseconds respectively, totaling 40 nanoseconds for T_{ccb} from recognition of the `WR_EN` signal.

The Timestamp delay, T_{tmp} , is 29 nanoseconds after the rising edge

of the WR_EN signal as described in Section 4.4. The two times for stability of the Event Record operate in parallel, therefore the Event Record stability is governed by Tccb, the greatest of the two delays. The Event Record for the first Event to be written is stable at the FIFO input 40 nanoseconds after recognition of the WR_EN signal. This means that at 16 MHz, the Write Controller state machine must wait 40 ns or 1 state time from recognition of the WR_EN signal to writing the Event Record into the FIFO buffer.

Analysis of the reset timing for the Write Controller deals with reset propagation delays through the Encoder, Event Register, and Decoder of the Write Controller and signal stability in the Event Logging and Timestamp Modules. The reset timing is broken into four areas of concern, which will determine the structure of the Write Controller state machine. The delays are Tcer (time from active clock event to active reset), Trce (time from active reset to active clock event), Trei (for time from active reset to a decision on the Event Indicator signal concerning the next Event), and finally Trwr (for the time from an active reset to the write of the second (or other) event). These four time constraints Tcer, Trce, Trei, and Trwr, along with the FIFO write setup times and pulse width times, will drive the state machine design.

The first time delay, T_{cer} , describes the minimum time which must elapse from an active clock event signal to an active reset signal. This covers the the Event Register delay and the setup time on the Decoder. This delay is 16.5 nanoseconds, meaning that one state time must occur between the activation of the Event Clock signal and the activation of the Event Reset.

The second time delay, T_{rce} , describes the minimum time which must elapse from an active reset signal to an active Clock Event. This time delay covers the time required for reset to propagate through the state machine PAL, the Decoder, the Event Logging module and the Encoder and still have a valid setup time for the Event Register. The minimum time between an active Reset_Event signal to an active Clock_Event signal is 141.5 nanoseconds. This value indicates that three state times must separate the active Reset_Event from the active Clock_Event.

The third timing constraint for design of the Write Controller state machine is T_{rei} , indicating the minimum time which must elapse from an active Reset_Event signal to a valid decision based on the Event_Indicator signal. This time delay accounts for settling of the Event indicators after an active reset, and includes the delay through the state machine, the Decoder, the Event Logging module, the Encoder and allows proper setup on the state machine for the Event_Indicator signal. This gives a total time from an active Reset_Event to a valid decision based on the

Event_Indicator input of 160 nanoseconds. This indicates that T_{rei} is three state times, or that three state times must separate the activation of the Reset_Event signal and a valid decision on the Event_Indicator signal.

The final time constraints for the state machine is T_{rwr} , for the time from an active Reset_Event to an active Write Signal from the state machine. This value indicates the time needed for Reset_Event to propagate through the state machine, the Decoder, the Event Logging module, then through the Command and Comment byte converters, and allow for a valid setup on the inputs to the FIFO memory. This gives the minimum time from an active Reset_Event to an active Write, T_{rwr} , as 206.5 nanoseconds. This value indicates that 4 state times must separate an active Reset_Event and an active Write for valid propagation delays and setup times.

The final constraints to discuss for the Write Controller state machine pertain to the width of the Clock_Event signal and the FIFO Write signal. The FIFO Write signal is specified for the Am7200-80 as 80 nanoseconds. This indicates that the Write signal pulse width, T_{ww} , must be two state times in width. The clock input to the 74F175 has a minimum width specification of 5 nanoseconds, indicating that the Clock_Event signal width, T_{cew} , must be at least 1 state time in width.

The two pulse width constraints are used to insure that the FIFO buffer and the Event Register are properly clocked. The four time delays, T_{wr} , T_{rei} , T_{rce} , and T_{cer} are 4 state times, 3 state times, 3 state times, and 1 state time respectively. These times form the design constraints needed for the Write Controller state machine, as it pertains to the actions for writing a second Event Record into the FIFO buffer. The times given previously for the FIFO write setup, or in particular T_{ccb} , is used to constrain the state machine state diagram for the writing of the first Event Record.

The state diagram for the Write Controller state machine is shown in Figure 4.14. The state machine given in Figure 4.14 implements the control needed within the time constraints listed previously. State S1 is the initial state for the machine and represents the start state. In state S1, the machine waits until the WR_EN signal is active, indicating the beginning of the microsecond.

If the $Event_Indicator$ is active, the machine makes a transition to state S3 and deactivates the $Event_Log_Enable$ (ELE in the state diagram); if no Event is active, the machine makes a transition to S2 and waits for WR_EN to deactivate before changing to S1 again. State S3 is used to simply allow the Event Indicators to stabilize. On the transition to state

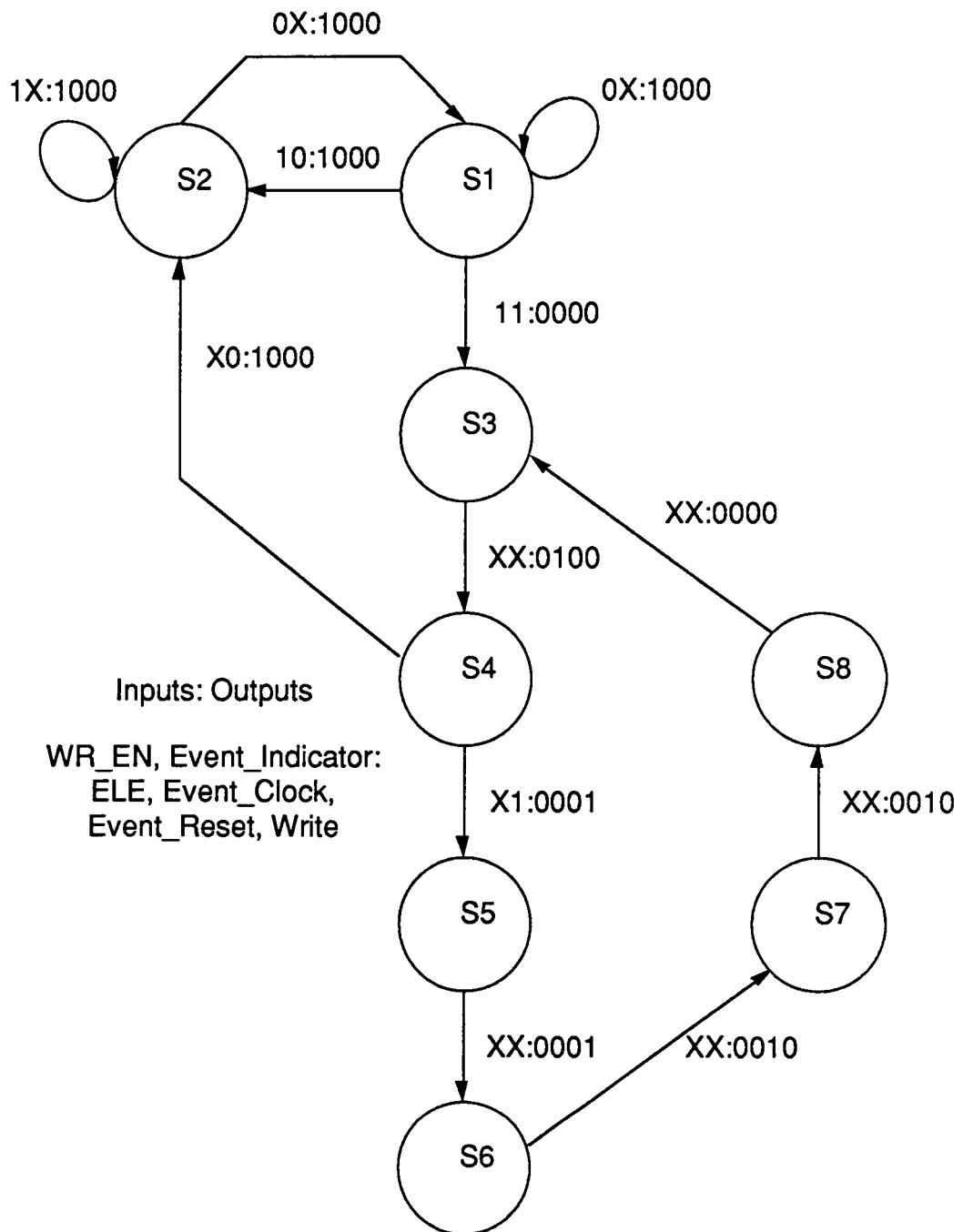


Figure 4.14: Write Controller State Diagram.

S4, the machine activates the Event_Clock signal and allows for setup of the Decoder. The transition from state S4 is used for decisions based on multiple Events. If the Event_Indicator signal is inactive, the machine transitions to state S2 to wait for deactivation of the WR_EN line. If the Event_Indicator is active, the transition from S4 is to S5, the Event_Clock is deactivated, the Write signal activated, writing the Event Record to the FIFO buffers. The change from state S5 to S6 is used to extend the width of the Write pulse for the FIFO memories. After state S6, the machine changes to state S7, and deactivates the Write signal and activate the Event_Reset signal. State S8 is used to extend the width of the Event_Reset pulse. After S8, the machine makes a transition to S3 with no output signals active. The S8 to S3 makes a transition allows the machine to look for the next Event, if applicable. The decision on whether a second Event is active is made in state S4 as explained above.

The machine described above meets the timing constraints given previously. Tccb is met between state S2 and S4, where two state time exists between deactivation of the Event_Log_Enable and activation of the Write signal for the first Event Record. Trwr is allowed for by the S6-S7-S8-S3 transitions. This allows for the 4 state times needed for proper Event Record setup after an Event Indicator Reset. Trei is allowed for by the S6-S7-S8-S3 transition as well, as state S4 is the state in which the decision concerning a second Event is made. Trce is given in the S6-S7-S8

transition of 3 state times which meets the specified constraint. T_{cer} , the time from Event_Clock activation to Event Reset, is met by state S3-S4-S5-S6 transition. The two pulse width constraints, T_{ww} and T_{cew} , are given by the S4-S5 transition and the S3 state.

With the state diagram given in Figure 4.14 the design of the Write Controller is complete. The Write Controller is capable of meeting the Record Rate of 2 Event Record writes per microsecond, set by IBM specifications. The Write Controller consists of a 74F148 priority encoder, a 74F175 register, a 74F138 Decoder, and a controlling state machine implemented into a PAL16R8. The encoder is used to distinguish different Event Indicators, the register is used to allow reset operation on only one Event Indicator, and the decoder is used to select Event Indicator resets. The state machine is used to control writing of Event Records into the FIFO buffer and control Event Indicator resets. The timing for the Write Controller has been analyzed and found to be sound. The Write Controller design is complete.

4.6 Small Computer Systems Interface

The Small Computer Systems Interface (SCSI) is introduced in Section 3.4 where the logical SCSI interface is discussed. This section discusses the implementation of the physical SCSI standard and includes a

derivation of the optimal SCSI packet size for the Monitor implementation. The optimal packet size derivation is used in Section 4.2 for determination of the FIFO buffer size.

4.6.1 SCSI Physical Implementation

The physical SCSI standard, for the eight bit SCSI bus (commonly called the SCSI-1 standard) is described by ANSI [1] to have nine data line and nine control lines. The nine data lines form an eight bit data bus with a parity bit, and the nine control lines are divided into phase lines, handshake signals, and global control signals. All lines are terminated with $220\ \Omega$ to VCC and $330\ \Omega$ to Ground, as outlined in Figure 4.15.

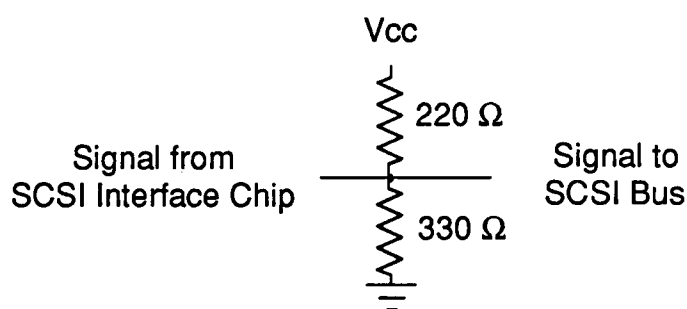


Figure 4.15: Line Termination for SCSI [1] .

To implement the timing outlined in the the ANSI SCSI-1 specification, several single chip options are available. For this project, the chip used was the Logic Devices 53C80. All the timing required for the

arbitration phase and the handshaking for asynchronous byte transfers are handled automatically, with an eight bit microprocessor interface for SCSI bus control, transmission and receipt. A DMA port is also available for use, allowing increased byte throughput.

Looking at Figure 4.1, the SCSI interface chip is interfaced to the DMA Controller and the microprocessor. The microprocessor is responsible for receiving all data, such as setup and control information from the PS/2 host and for controlling the data transmission to the host system. Under control of the microprocessor, the DMA controller is responsible for sending Event Records to the host.

The SCSI interface implementation is shown in Figure 4.16, consists of SCSI interface, Termination, Address Decoding, and Interrupt Interface.

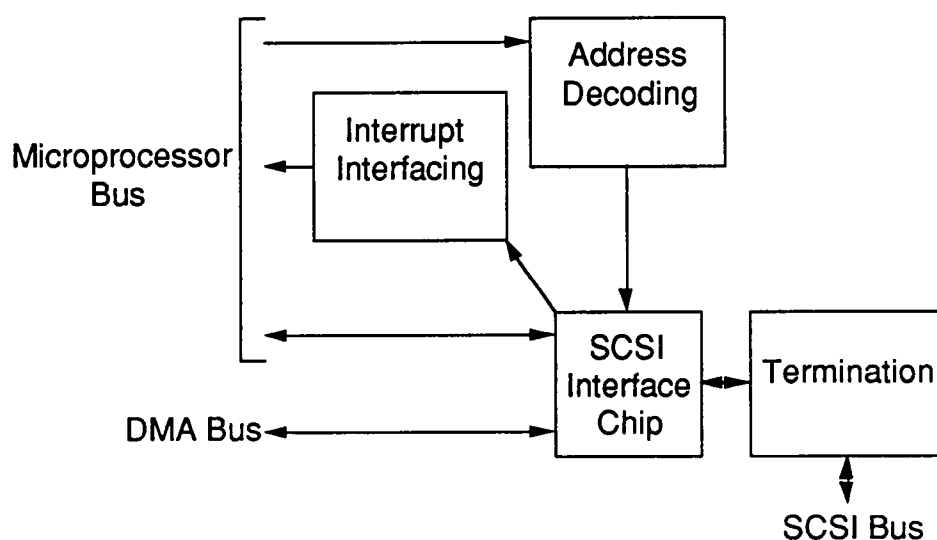


Figure 4.16: Block Diagram for the SCSI Interface.

Interrupt Interfacing is used with the microprocessor to allow the Logic Device chip to operate in an interrupt driven mode of operation, freeing the microprocessor from having to poll the chip for information. The Interrupt Interfacing is also needed to allow other devices access to the interrupt line of the microprocessor. The address decoding is used to place the SCSI interface chip into the memory map of the microprocessor, allowing easier access to the registers in the 53C80 chip. The termination and the SCSI interface chip, the 53C80, are discussed above.

4.6.2 SCSI Data Packet Size Derivation

The SCSI transfer rate and the byte accumulation rate will be analyzed to derive the minimum length of the SCSI data phase. This value is also the length-width product for the FIFO buffers.

As discussed in Section 3.4.2, a SCSI bus transfer consists of three basic phases: ARBITRATION/SELECTION, COMMAND, and DATA phases. The ARBITRATION/SELECTION and COMMAND phases can be considered overhead for three reasons; these phases do not directly contribute to data transfer, both these phases are under the control of the Microprocessor and are therefore slow, and these phase lengths (times and byte lengths) are not under design control. The DATA phase length, on the

other hand, contributes directly to data transfer, is under design control, and the SCSI data transfer occurs through the use of a DMA controller, which is much faster than Microprocessor control. To initiate a SCSI transfer for each Event Record would be impractical, due to the overhead introduced by the ARBITRATION/SELECTION and COMMAND phases. Therefore, the length of the DATA phase should be increased to as large a size as possible, reducing the percentage overhead caused by the ARBITRATION/SELECTION and COMMAND phases. However the length of the DATA phase is limited by storage requirements.

The ARBITRATION/SELECTION times are set by the SCSI specification and can not be altered. The COMMAND phase length is set at 6 bytes due to software limitations on the PS/2. Experimental results show that the ARBITRATION/SELECTION and COMMAND phases of the SCSI transfer take approximately 600 microseconds, which, as explained previously, forms overhead time for each data transfer. The maximum byte rate for the system is calculated in Section 3.3 to be 125 kilobytes/second.

To determine the size of the SCSI data packet, two equations need to be derived as functions of the data packet size. These equations represent accumulation time per packet and the transfer time per packet. The break even point on data packet length occurs when the accumulation time per packet and the transfer time per packet are equal. At this point, the FIFOs

are storing and sending data at an equal rate. By setting the accumulation time per packet and the transfer time per packet equal and solving for the data packet size, the minimum allowable packet size is derived.

The accumulation time per packet is a function of bytes/data packet divided by the maximum byte rate given in Section 3.3. This equation is

$$\frac{\text{accumulation time}}{\text{packet}} = \frac{\text{bytes}}{\text{packet}} \frac{1}{\text{byte rate}} \quad (4.1)$$

If X represents the bytes/packet, and T1 represents the accumulation time/packet, the equation 4.1 becomes equation 4.2, with the byte rate given in section 3.3.

$$T1 = X \left(\frac{1\text{second}}{125 \text{ Kbytes}} \right) \quad (4.2)$$

In Equation 4.2, T1 represents the time needed to accumulate X bytes for the formation of one data packet to be sent over the SCSI bus.

The equation for the transfer time per packet is more complicated. The transfer time consists of two parts, the time needed to complete the ARBITRATION/SELECTION and COMMAND phases, and the time needed to transfer the data. As stated previously, the Arbitration and Command information is sent by the monitor Microprocessor, and the Data information is sent by a DMA controller. Therefore, the two sets of

information are sent at different byte rates. The Arbitration and Command information, representing overhead, takes 600 microseconds per packet to transfer, and is the same for every packet sent. The Data information depends on the amount of data sent and the data transfer rate. The data rate, controlled by the DMA Controller, is set at 1 byte/microsecond, a rate easily achievable by the DMA Controller and sufficient for the data transfer needs. If T_2 represents the transfer time per packet, and X once again represents the amount of data in each packet, then Equation 4.3 describes T_2 as a function of X .

$$T_2 = 600 \mu\text{seconds} + X \left(\frac{1 \mu\text{second}}{\text{byte}} \right) \quad (4.3)$$

In Equation 4.3, T_2 represents the time needed to transfer X bytes over the SCSI bus and represents the two data transfer requirements. Rewriting equation 4.2 to change the units, equation 4.4 results. By setting equation 4.3 equal to 4.4, X can be solved for.

$$T_1 = X \left(\frac{8 \mu\text{seconds}}{\text{byte}} \right) \quad (4.4)$$

Equation 4.5 represents Equation 4.4 set equal to equation 4.3.

$$600 \mu\text{seconds} + X \left(\frac{1 \mu\text{second}}{\text{byte}} \right) = X \left(\frac{8 \mu\text{seconds}}{\text{byte}} \right) \quad (4.5)$$

Solving equation 4.5 for X results in equation 4.6.

$$X = \frac{600 \mu\text{seconds}/\text{packet}}{7 \mu\text{seconds}/\text{byte}} = 85.71 \text{ bytes}/\text{packet} \quad (4.6)$$

The units of microseconds cancel leaving X with a value of 85.71 bytes/packet. The value of X represents the minimum number of bytes per packet allowable, with the monitor still able to transfer data to the PS/2 at the maximum record rate of 125 kilobytes/second or maximum Event Rate of 31062 Events/second. If the Event Rate is at its maximum, and the bytes per packet is 85.71, then the monitor is logging recording events at the same rate as it is transferring Event Records. Increasing the number of bytes/packet reduces the percentage overhead of the ARBITRATION/SELECTION and COMMAND phases and improves transfer characteristics. The number of data bytes sent with each SCSI packet is set at 512, well above the derived minimum of 85.71 bytes/packet, and this value of storage is easily obtainable with many common FIFO products.

4.7 DMA Controller

The DMA Controller is an interface between the FIFO memory and the SCSI interface chip thereby speeding transfer of Event Records from the FIFO Buffer to the SCSI bus. The Microprocessor sets up the SCSI interface chip for a DMA transfer, starts the DMA Controller, then relinquishes control of the data bus. When the DMA transfer is complete, the DMA controller is responsible for interrupting the Microprocessor, which ends the transfer of one packet of Event Records. A global view of the DMA Controller is shown in Figure 4.1, and Figure 4.17 shows the interface signals for the DMA port on the SCSI Interface, the Read signal for the FIFO buffers, and the start signal from the microprocessor.

The data transfer rate for the DMA controller is set in Section 4.2 at 1 byte every microsecond. The DMA Controller takes two major steps. First is to read the FIFO, thereby driving the data bus with the value to be written to the SCSI bus, then the data is written to the DMA port on the SCSI Interface. Using this method the data is not stored in the DMA controller, eliminating a write step in a DMA byte transfer.

To handle data transfer responsibility the DMA Controller has interfaces to the Microprocessor, the 53C80 DMA port and the FIFO read control. The SCSI Interface chip, the Logic Devices 53C80, has the most complicated interface with the DMA Controller. The four signals, /DRQ,

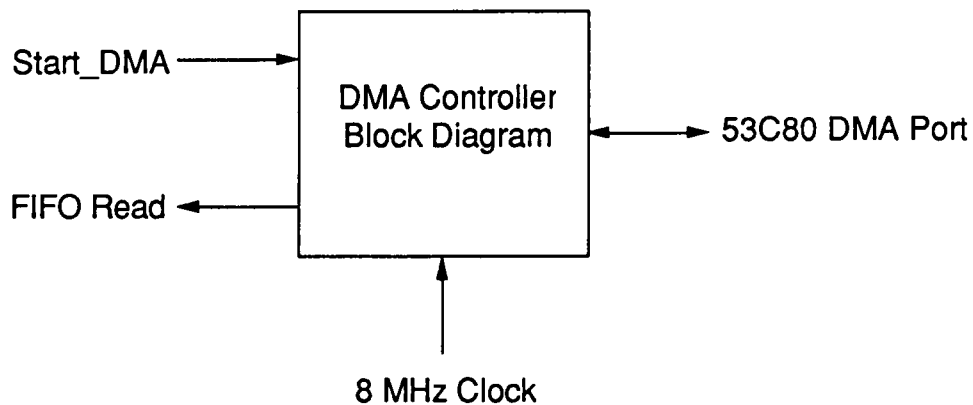


Figure 4.17: Overall View of the DMA Controller. The 53C80 DMA signals and FIFO Read signals shown.

$\overline{\text{DACK}}$, $\overline{\text{IOW}}$ and $\overline{\text{EOP}}$ control DMA data flow to the SCSI interface chip. The signal $\overline{\text{DRQ}}$ is the Data Request signal and is an output of the 53C80. The signals $\overline{\text{DACK}}$ and $\overline{\text{IOW}}$, both inputs to the 53C80, are used to write data. The $\overline{\text{EOP}}$ (End Of Process) signal is an input to the 53C80 which is used to end a DMA transfer. This signal is used to cause an interrupt to the Microprocessor, indicating that the DMA transfer has ended.

The FIFO buffer, comprised of four separate byte-wide FIFO chips, has four read signals, $\overline{\text{R0}}\text{-}\overline{\text{R3}}$, that are controlled by the DMA controller. When a read signal is active, the individual FIFO will drive the next byte in its buffer onto the data bus. The read signals form the FIFO interface to the DMA Controller. The Microprocessor has two interface signals to the DMA controller, Start_DMA and $\overline{\text{RESET}}$, used for overall control of the DMA process.

Timing for the DMA and FIFO ports are of great importance in determining control for the DMA controller. The Microprocessor signal, Start_DMA, is used only to start the DMA transfer and is considered both asynchronous to the DMA Controller and slowly changing, therefore its timing will not be considered. The timing for the DMA port of the 53C80 will be considered first, followed by the timing of the FIFO buffers.

The DMA controller uses the DMA port on the SCSI interface chip for writing data to the 53C80 in Initiator Mode. The Timing Diagram for the DMA port on the 53C80 is shown in Figure 4.18 and Table 4.1 [7].

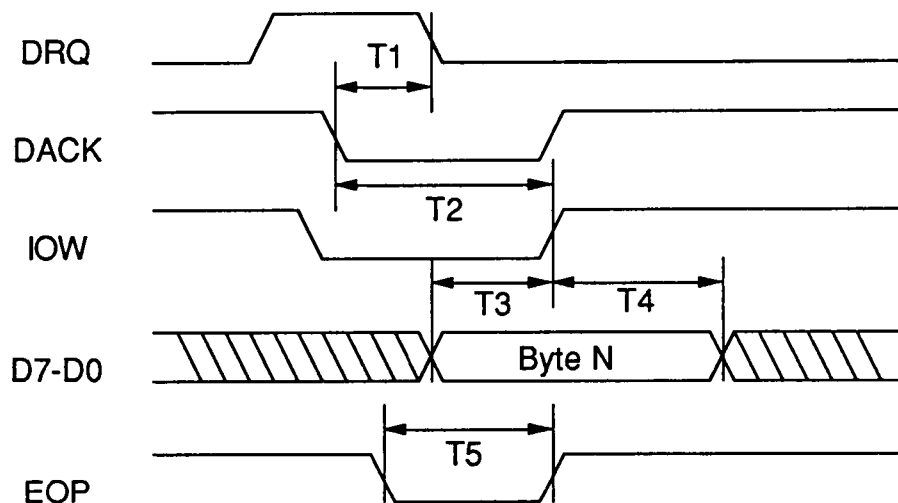


Figure 4.18: DMA Initiator Write Timing for the 53C80.

Table 4.1: Timing Values for Figure 4.18. All values are in nanoseconds.

		<u>Min</u>	<u>Max</u>
T1	DRQ false from Write EN(IOW and DACK)	-	60
T2	Width of Write Enable	60	-
T3	Data Setup to end of Write Enable	20	-
T4	Data Hold from End of Write Enable	15	-
T5	Concurrent Width of EOP, IOW, DACK	50	-

The DMA port interface on the 53C80 is simple, consisting of only four signals and the data bus. The three signals DRQ, /DACK, /IOW interact to handshake bytes to the 53C80. The signal /EOP is used to indicate when the final byte is transferred from the FIFO buffer to the SCSI Interface. As indicated in Figure 4.18, DRQ, an output from the 53C80, is the start signal for each byte of the transfer. In response, /DACK and /IOW should activate (both do not have to activate together, but must be active together for the specified time, T2). The occurrence of /DACK and /IOW together is labeled as Write Enable. Once Write Enable is active, DRQ will deactivate in T1 time. The rising edge of /DACK or /IOW indicates a write to the 53C80, therefore the data must be set up and held for T3 and T4 time respectively before and after the end of the Write Enable. On the last byte of the DMA transfer, /EOP must activate concurrent with active /IOW and /DACK for T6 time. This indicates the last byte to the 53C80 and causes an interrupt to the microprocessor.

The read timing for the FIFO memory must also be met by the DMA Controller. The read timing for the Am7200-80 FIFO Memory is shown in Figure 4.19 and the timing values are given in Table 4.2.

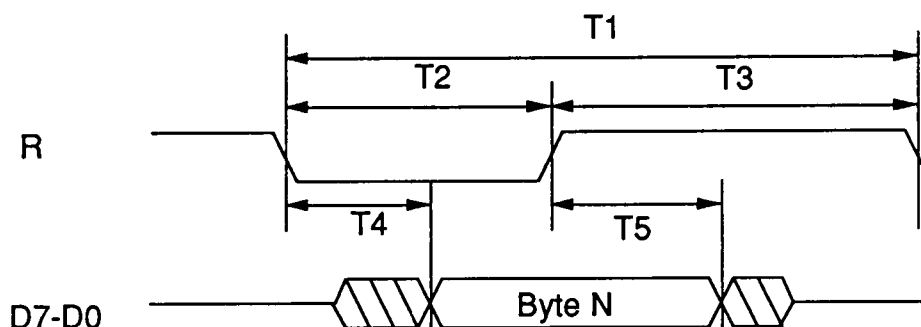


Figure 4.19: Read Timing for the Am7200-80 FIFO.

Table 4.2: Timing Values for Figure 4.19. All values are in nanosecond.

		<u>Min</u>	<u>Max</u>
T1	Read Cycle Time	100	-
T2	Read Pulse Width	80	-
T3	Read Recovery Time	20	-
T4	Access Time	-	80
T5	Data Valid after inactive Read Signal	5	-

Reading data from the FIFO memory consists of manipulating only one signal, the read signal "R". The Read Cycle Time, T1, sets the minimum time which the FIFO can be reread at 100 nanoseconds. The Read Pulse Width, T2, is the minimum time which the Read signal can be

active, and is specified as 80 nanoseconds. The Read Recovery Time, T3, is 20 nanoseconds. Data is valid from the Am7200-80, T4 time, or 80 nanoseconds, after an active Read signal "R". Data is valid for 5 nanoseconds after an inactive Read signal. This timing coupled with the Timing for the DMA port on the 53C80, given in Figure 4.18 and Table 4.4, must be used to design the DMA Controller.

The design for the DMA Controller consists of two sections, layout of the DMA Controller architecture and layout of the control needed to accomplish the DMA Controller tasks. A written description of the DMA Controller actions will be given first, followed by a layout of the DMA Controller hardware and a description of the control needed. To start the process, the microprocessor sets the 53C80 for DMA operations, then activates the Start_DMA signal. When the 53C80 has been set up for DMA operations and the chip is ready for a byte, the DRQ signal is activated. The DMA Controller must watch for both the DRQ signal and the Start_DMA signal to become active. Once these signals are active, the DMA Controller activates a read signal on the FIFO buffer and writes the byte to the SCSI Interface by activating the IOW and DACK lines simultaneously. When the IOW and DACK lines make a transition from an active state to an inactive state the byte is written to the 5380.

When the DRQ line is active again, the transfer starts again. The DMA controller counts the bytes transferred and as the final byte is written, the /EOP line is activated at the proper time, indicating DMA transfer end.

The above paragraph verbally expressed the functional needs for the DMA Controller. It must handle the interface between the FIFO memory and the DMA port on the 53C80 within the timing specifications. It must be able to translate which FIFO device it is reading from. It must be able to count the total number of bytes sent and it must react to the final byte, to activate the EOP signal. The DMA Controller needs a counter which can count as high as 512 bytes, as this is the packet size derived in Section 3.2. The Controller needs a decoder to select the FIFO and needs a state machine for control. Figure 4.20 shows the DMA Controller hardware.

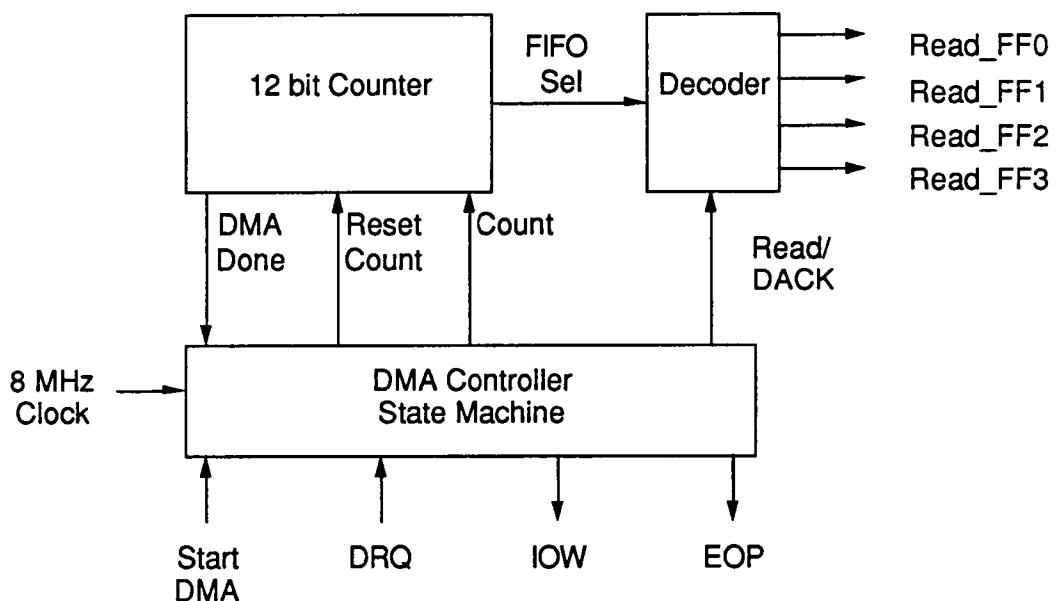


Figure 4.20: Block Diagram for the DMA Controller.

The decoder used is a 74LS138, a 3 to 8 TTL decoder and the counter consists of three 74LS161A parts. The state machine is implemented in a PAL16R8B with a 15 nanosecond propagation delay.

The 74LS138 3 to 8 decoder has three inputs for determining the active output line and has an enable input. The decoder has eight active low outputs, with one output active when the decoder is enabled corresponding to the value on the inputs. For the DMA Controller the decoder is used to determine one of four FIFO chips. The enable input is connected to the signal Read from the DMA Controller state machine. The Read signal performs as the only enable signal. The decoder inputs are tied to the DMA counter. The outputs are tied to the FIFO read signals.

The counter for the DMA Controller has two major purposes, to determine which of the FIFOs is currently being read and to determine when an entire packet has been sent, allowing activation of the EOP signal. The lower two bits of the counter are connected to the 74LS138 decoder allowing FIFO read signal selection. To determine when the DMA packet has been sent, bit 10 of the counter forms the DMA_Done signal.

The state diagram for the DMA Controller state machine is shown in Figure 4.21 below. The state diagram will be explained to verify that the DMA Controller meets timing specifications.

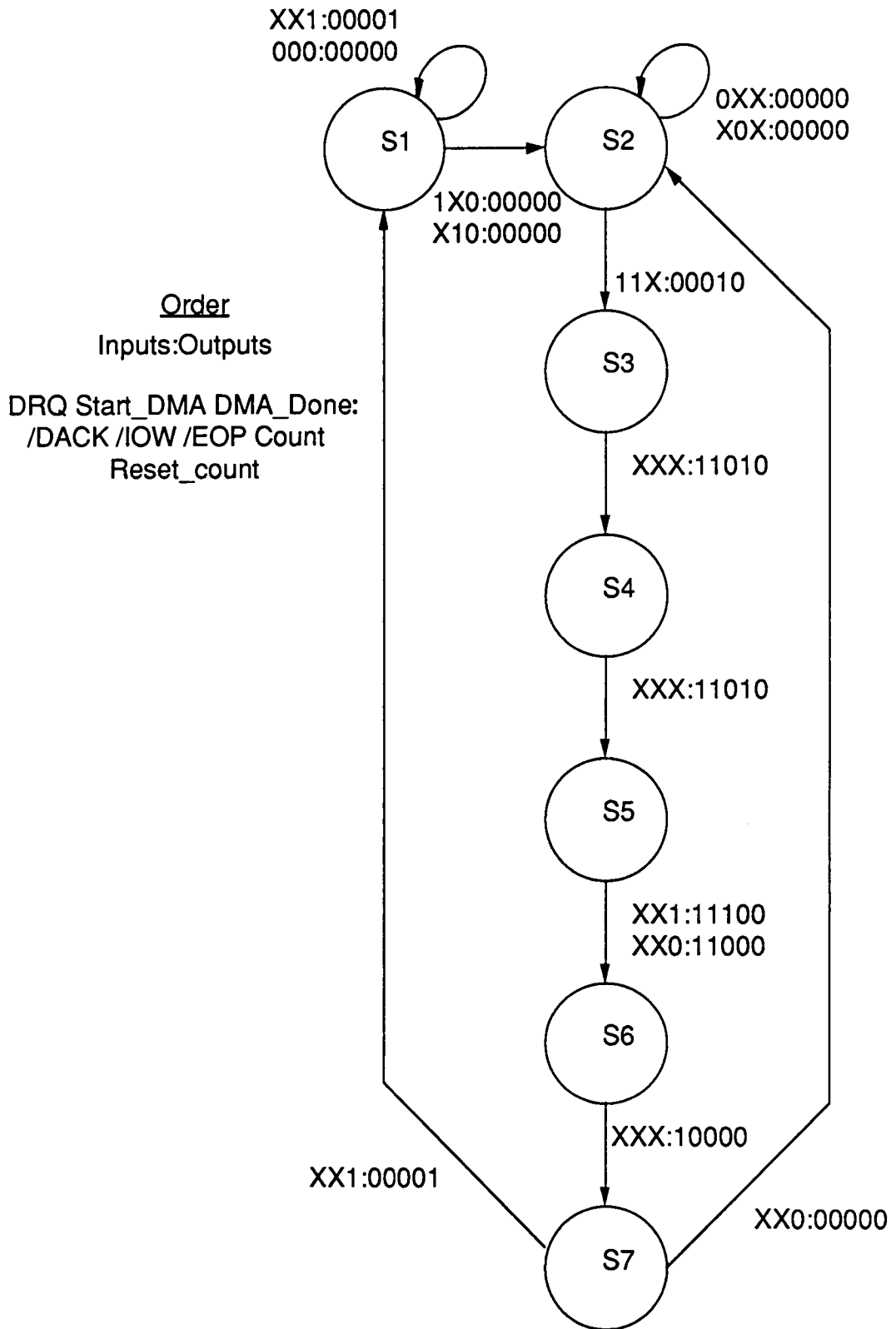


Figure 4.21: State Diagram for the DMA Controller State Machine.

The clock for the state machine is 8MHz, a value picked because it minimizes the number of states needed in the DMA Controller state machine (as will be shown) and because this clock frequency is readily available.

The DMA Controller state machine consists of seven states, as seen in Figure 4.21. The start state is S1. The transition from state S1 to S2 is accomplished when either DRQ is active or Start_DMA from the microprocessor is active. State S2 is a hold state until both Start_DMA and DRQ are active, indicating a valid start, and causing the machine to make a transition to state S3. When the transition to S3 is made, the machine activates the Clock line to the DMA counter, incrementing the counter. The transition from state S3 to S4 is in place to allow the required amount of time for the data to be read from the proper FIFO and setup for the write into the DMA port on the 53C80. State S5 is in place to allow for a decision to be made concerning the /EOP signal, and state S6 is used to allow for proper hold time on the Data while at the same time, writing the data into the 53C80 (through the deactivation of the /IOW signal). State S7 is used to decide whether the DMA Controller has sent the required number of bytes to the 53C80. If so, the state machine jumps to the start state and waits for the next start signal. If not, then the state machine makes a transitions to state S2 and writes the next byte.

A timing analysis of the DMA Controller state machine, to verify that timing specifications are met, follows. Four steps of the timing analysis will be analyzed, counter value setup for read, read time and 53C80 setup, /EOP window width, and 53C80 hold time. These times fulfill the major timing requirements for the DMA Controller machine.

The counter setup delay is the time it takes the Count signal to propagate from the DMA Controller State Machine to the output of the 74LS138 decoder. This delay totals of 109 nanoseconds for the set up time for both the reading of the FIFOs and the decision for /EOP activation. This time is less than one state time of 125 nanoseconds with an 8 MHz clock, therefore one state time is sufficient for counter setup before the read operation needed in the S2 to S3 transition. The /EOP decision is three state times away from the activation of the Count signal.

The time required to read the FIFO memory and setup the 53C80 governs the transitions from state S3 to state S6. This time is 154 nanoseconds, indicates that two state times are needed for the Read activation to Data setup time. The transition from S3 to S6 is adequate.

The activation of /EOP occurs if needed in state S6. The /EOP signal must be active with /IOW and /DACK for 50 nanoseconds to indicate a valid End Of Process to the 53C80 and to allow an interrupt to end the DMA transfer. This indicates that one state time of 125 nanoseconds is sufficient for the activation of the /EOP signal.

This concludes the timing analysis for the DMA controller state machine, and finishes coverage of all major timing concerns. The hardware for the DMA Controller has been partitioned and described, with interfacing signals between the DMA Controller blocks and external to the DMA Controller defined. The DMA Controller state machine state diagram has been given and described, and the timing for the FIFO memory and the 53C80 with respect to the DMA controller has been verified.

4.8 Microprocessor

This section discusses the Microprocessor functions in the monitor hardware. The Microprocessor in the monitor hardware is used for overall control of the functions of the monitor and for communication with the host processor for command downloads and data uploads. The monitor control consists of two sections, control of hardware reset and control of Event captures. The communication control consists of interfacing with the SCSI interface chip, the 53C80, for receipt of commands as well as initial contact with the host. It is also responsible for control of the DMA process when sending Event Records.

The microprocessor has two areas which it controls. The first, hardware reset control, allows the microprocessor to reset all the capture logic, DMA Controller, FIFO buffers, etc. without entering reset itself. This control eases testing of hardware and firmware debugging and allows the microprocessor to maintain the hardware reset as a means of stopping Events from being logged. The 53C80 is not connected to the reset signal for the other hardware, allowing host communications to also remain untouched when testing the hardware or while not capturing Events. The 53C80 and the microprocessor are tied to a common reset signal for these two chips only, and is available to the user for an overall reset feature.

Secondly, the microprocessor exerts event capture control over the hardware. This is accomplished using two signals, Capture_Levels and Capture_Waits, which control the Level And Wait Event Capture modules respectively. The microprocessor can set these signals to capture any combination of Level and Wait Events. Notice that the Timer Overflows are always captured (whenever the hardware reset is inactive) and is not under the control of the microprocessor. Timer Overflows represent a basic timestamp feature, and the other Event Records do not mean much without them.

Communication control is accomplished using the microprocessor and the 53C80 SCSI Interface Chip. The communication is handled in two sections, command reception from the host and Event Record upload to the

host. The 53C80 can interrupt the microprocessor, and command reception from the host is handled using the 53C80 interrupt capability. The interrupt service routine simply receives the entire command, stores it in memory, and sets a flag indicating that the command is available. The main loop for the microprocessor is responsible for watching this flag and for reacting to the commands sent. The commands sent to the monitor are covered in Section 3.5, which covers the host control of the monitor hardware. All six of the commands listed are designed to reduce the time needed to accomplish the task. This frees the main loop from having to perform time intensive tasks, and allows proper monitoring for commands received from the host via the 53C80.

As the second part of control of communications, the microprocessor has high level control of the DMA transfer of Event Records to the host. This control is accomplished as a two stage process. The main loop of the microprocessor code simply polls a signal from the FIFO memory, the Half-Full signal, which indicates that the DMA transfer should occur to prevent a FIFO buffer overflow. When this signal, `DMA_Needed`, goes active, the microprocessor simply contacts the host, with the host in Target Mode and the monitor in Initiator mode and sends the command to receive Event Records in 512 byte packets. Once the microprocessor has handled the Arbitration, Selection, and Command phases of the SCSI protocol, it watches for the Data In Phase (the phase signals are set by the Target), and

then sets the 53C80 for DMA Initiator Send mode. Once the 53C80 is setup, the microprocessor activates the DMA_Start signal, allowing the DMA Controller to begin transferring the bytes of data from the FIFO buffers to the DMA port on the 53C80. The microprocessor now enters a wait state until it is interrupted. When the DMA controller has finished its 512 transfers, the /EOP signal is activated to the DMA port on the 53C80 and the SCSI Interface Chips ending the DMA transfer, and interrupts the microprocessor. The microprocessor finishes the SCSI transfer by sending the status byte and watches for the Bus Free phase. The transfer of Event Records to the host is accomplished and the microprocessor continues.

The firmware for the microprocessor is divided into three sections, the main loop, the DMA transfer control, and the SCSI interrupt service. The main loop is responsible for two major tasks: monitoring the DMA_Needed signal and responding to commands sent from the host. The SCSI Interrupt service is responsible for correctly communicating with the host in Target mode and placing commands in a command buffer. The DMA transfer control must handle communication with the host in Initiator mode and implement the DMA control described above. These three software sections form the firmware for the microprocessor.

The tasks above do not indicate a great need for speed or processing power on the part of the microprocessor. Its task is simply monitoring and controlling three operations: the SCSI communication, command

processing, and DMA transfer. Since none of these tasks is speed or processing intensive, the microprocessor needed does not need to be complicated or fast. Thus the microprocessor chosen was the Motorola 68HC11 running at 2 MHz and using a FORTH Kernel for an operating system and programming language. This microcontroller is readily available for use in the Electrical and Computer Engineering Department of the University of Tennessee, and several of the Professors have had experience with this microprocessor. The main loop was written in FORTH, while the SCSI interrupt service routine and DMA control were both written in assembly language. The M68HC11 has more features than were needed for the monitor hardware control but its availability and the availability of a large base of support and previous use caused the choice of this processor.

CHAPTER 5

SUMMARY AND CONCLUSIONS

This chapter presents a summary for the Priority Event Monitor architecture and design. The present design is critiqued and future alternatives and changes are suggested. Presented first is a summary of a working version of the current design. This is followed by suggested architectural changes to the Priority Event Monitor, that could improve the performance of the system and could reduce the cost. Following this are design and implementation changes suggested for the Monitor.

5.1 Results of Present Implementation

The present design has been implemented and is working. The record rate of 2 Events each microsecond is met with the Write Controller operating at the design value of 16 MHz. The Peak Rate of 32 Events in 32 μ seconds is met through the use of the 256 by 4 byte FIFO buffer. The design is stable and offers comparable performance to the previous system used by IBM, at reduced cost and reduced chip count and board space.

The only specification not met with the present implementation is the Event Throughput rate of 31060 Events/Second. The limitation of the system did not rest with the External Monitor hardware, which was designed to sustain the needed Event Rate. The limitation rests with the

processor speed of the host computer and with the software used to interface to the External Monitor. The present implementation uses an IBM PS/2 model 50Z, with a 80286 processor. A maximum throughput of 5000 Events/Second was reached, which encompasses the general operating range shown in Graph B.2. Tests were performed using a PS/2 Model 70, with an Intel 80386 processor and a maximum Event Rate of 30000 Events/Second was met and was deemed acceptable. The current implementation thus retains the PS/2 Model 50Z as the Host and if Event Rates or greater than 5000 Events/Second are needed, the present SCSI card will have to be installed into a faster PS/2 system.

The SCSI bus proved to be a workable instrument interface bus. Most existing uses of the SCSI bus have been for peripheral connection to computer systems, such as hard disks and CD-ROMs to a central CPU. The SCSI bus offers a faster alternative to busses such as the IEEE 488 bus when large amounts of data need to be transferred from instrument to host.

The present Priority Event Monitor consists of a PS/2 Model 50Z as host, a physically separate Monitor for 4702 observation and a SCSI bus for connections. The present design works and is acceptable for most Event Rates under observation.

5.2 Suggested Priority Event Monitor Improvements

This section discusses several changes that could be made to the Priority Event Monitor to improve performance and to reduce the system complexity. The changes discussed include changes to both the Priority Event Monitor architecture and to the monitor hardware architecture.

5.2.1 Architectural Change

The present architecture for the Priority Event Monitor was found to be acceptable in implementing the specifications given. The design is stable and working, though the complexity of the system was found at times to be greater than needed. Debugging the DMA Controller, SCSI Interface and the PS/2 Host interactions proved difficult. Also, the present architecture has the Monitor as a separate physical entity from the PS/2 host requiring a communication connection and more space for setup. One architectural change is suggested to reduce interaction complexity and reduce the physical complexity of the system.

The architectural change eliminates the separate Monitor and the SCSI bus by moving the Monitor hardware onto one PS/2 Microchannel card. This architectural change has the benefit of reducing complexity by using available PS/2 resources and by reducing physical space needed. The

PS/2 CPU would function as the Priority Event Monitor controller and the existing PS/2 DMA controller would handle data transfers from the FIFO buffers to main memory. Both data transfers and commands between the Monitor to the PS/2 main memory would be faster when compared to the SCSI bus. The architectural change is shown in Figure 5.1.

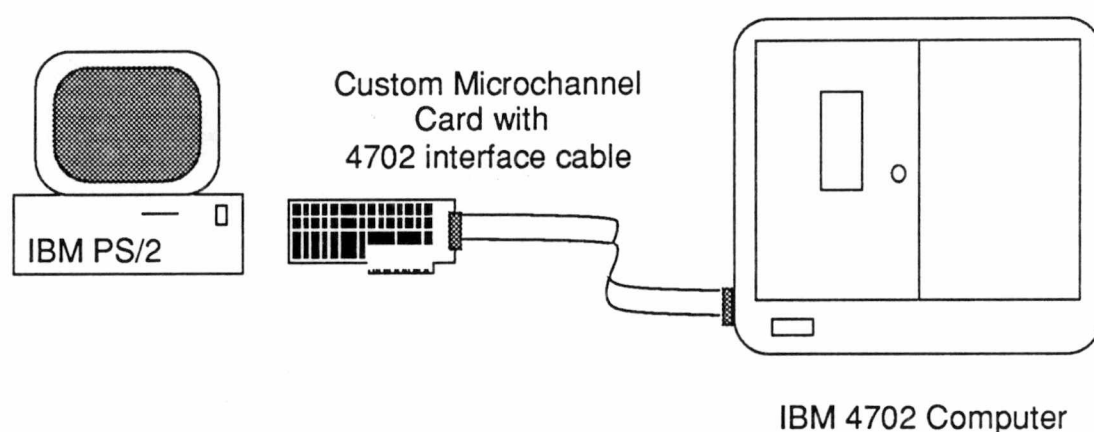


Figure 5.1: Improved Priority Event Monitor Architecture.

The architectural effects of the change are (1) Elimination of the SCSI bus adapter for the PS/2, (2) Elimination of the SCSI bus hardware for the Monitor. (3) Consolidation of CPU functions to one system, the PS/2, (4) Consolidation of DMA and memory transfer control, (5) Increased PS/2 software. The elimination of the SCSI bus from the PS/2 and the Monitor simply removed one layer of communication between the two, simplifying information transfer. Communication is faster due the the use of the Microchannel bus. Consolidation of CPU and DMA functions

reduces design and implementation complexity of the system. The increase in PS/2 software is trivial compared to reduced system complexity and increase in system speed. The architectural changes are reflected in the Monitor hardware architecture as shown in the next section.

5.2.2 Monitor Hardware Changes

This section discusses three changes suggested for the Monitor. The first change discussed relates the architecture improvements suggested above to the Monitor architecture. The second change involves reduction of the timer length to 8 bits, and the third change involves using asynchronous, or level-mode, logic for the Event Capture module. The change in Monitor architecture and in the length of the timer length results in reduced hardware and reduced complexity. The use of asynchronous logic for the Event Capture module increases the speed at which the Write Controller can operate and increases system speed.

Reduction of the Monitor hardware to a single PS/2 card changes the Monitor architecture by (1) eliminating the SCSI bus interface, (2) eliminating the CPU, (3) eliminating the DMA Controller, and (4) adding a Microchannel interface. These changes yield a significantly different Monitor architecture from that shown in Figure 4.1. The improved Monitor architecture is shown in Figure 5.2 below.

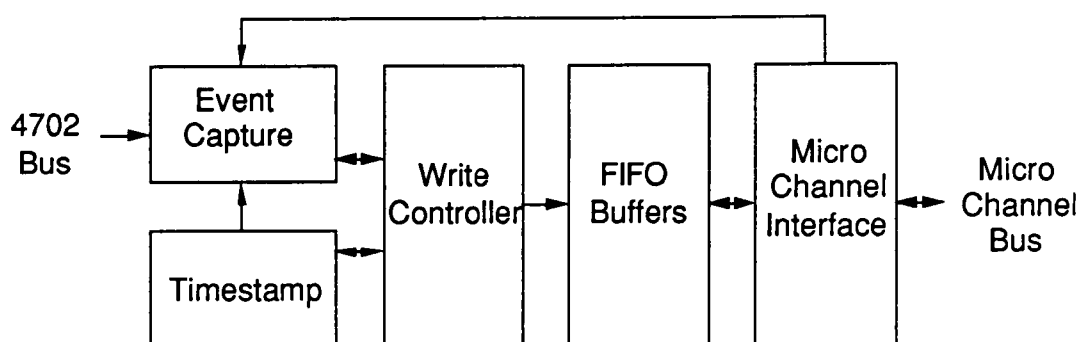


Figure 5.2: Improved Monitor Architecture.

The elimination of the microprocessor, custom DMA Controller, and SCSI interface for the monitor hardware does have a cost to be considered. A Microchannel interface must be built to take the place of the SCSI bus. Device Driver code must also be written for the PS/2 host for capture control and for Event Record uploads, and registers would have to be added to control the Event Capture modules. These additions, while significant, are outweighed by the reduced complexity and hardware overall.

The second suggested change to the Monitor involves reduction of the Time field in the Event Records to 1 byte. This change would reduce the maximum byte rate from 125061 bytes per second for 31250 Events per second to 105469 bytes per second. In addition, reducing the Time field reduces FIFO Buffer width and reduces the timer hardware needed. The length of the Time field for the Event Record should have been made on the basis of maximum operating speed, not typical operating conditions.

The final suggested change concerns the design and implementation of the Event Capture module on the Monitor hardware. The present design for the Event Capture module uses a synchronous, clocked, state machine to detect the state of the 4702 external busses. The use of a synchronous machine yields an inherent latency due the clock frequency in recognizing Events and accepting Event resets from the Write Controller. The use of an asynchronous, or level-mode, machine would reduce the Event recognition time, reduce the Event reset time and could increase the speed of the Write Controller from 2 Event writes each μsecond to 3 or more Events per μsecond . The use of an asynchronous machine would require a larger size machine and a greater hardware implementation, but these negative effects could be offset through greater integration of the Monitor, as will be discussed in the next section.

5.3 Suggested Implementation Changes

Several suggestions are presented in the previous sections concerning architectural and design changes to the Priority Event Monitor and to the Monitor hardware. This section outlines implementation changes that could also reduce the complexity of the Monitor hardware and make the previously mentions architectural changes a greater possibility.

Implementation improvements could be made to the Write Controller module through integration of the various parts onto fewer chips. The entire Write Controller could possibly be implemented using a single PAL22V10, instead of using a PAL16R8, a 74F148 encoder, a 74F175 register, and a 74F138 decoder. Preliminary work in this area indicates that this is feasible. This one change would decrease the board space use, help in reducing debugging time, and help increase the speed of the Write Controller.

Field Programmable Gate Arrays could be used to implement the Event Capture Module, the Timestamp Module, and the Write Controller into one or two chips, significantly reducing the chip count for the front end of the Monitor hardware. By implementing the architecture changes and integrating several of the modules the Monitor would consist of a handful of chips. One or two chips for the front end capture, timestamp, and write circuits, four FIFO chips, the Microchannel interface chips, and some glue logic to interface the FIFO buffers to the Microchannel. A truly integrated system would then be accomplished and this system would easily fit onto a single Microchannel card.

LIST OF REFERENCES

- [1] American National Standard for Information Systems., X3.131-1986 Small Computer Systems Interface, June 23, 1986.
- [2] Ciarcia, Steve, "Adding SCSI to the SB100 Computer Part 1: Introduction," Byte Magazine, pp. 85-94, May 1986.
- [3] Ciarcia, Steve, "Adding SCSI to the SB100 Computer Part 2: Bus Phases," Byte Magazine, pp. 107-114, June 1986.
- [4] Skinner, David, "Speed Storage Data Transfers with the new SCSI Standard," Electronic Design, July 14, 1988.
- [5] Advanced Micro Devices, "Introduction to First-In First-Out Memories", CMOS Memory Products Data Book/Handbook, 1991.
- [6] American National Standard for Information Systems/Institute for Electrical and Electronics Engineers., Standard 488.1-1987 Digital Interface for Programmable Instrumentation, 1988.
- [7] Logic Devices Incorporated, "CMOS SCSI Bus Controller L5380/L53C80", Fast CMOS Databook, 1989.
- [8] Texas Instruments Incorporated, F Logic (SN54/74F) Data Book, 1989

APPENDICES

APPENDIX A

ORIGINAL PROJECT SPECIFICATIONS

The original project specification are given below. Please note that the document listed next is an exact copy of the original specifications, as listed by IBM. The document gives several suggestions as to the structure of the software and the hardware needed for the Priority Event Monitor, some of which was not used. Also, since the terminology used in the body of this work was derived over the course of time, the terminology used in the specification document differs somewhat from the main body. The specification is listed beginning with the next page.

**UNIVERSITY AGREEMENT #86-115
APPENDIX I
Spring Quarter, 1990**

November, 1989

Lloyd Conard

IBM Corporation
Branch Delivery Systems
Dept. 58J / Bldg. 202
1001 Harris Blvd
Charlotte, NC 28257

Objectives

Design, build, code, and debug a PC Based 4700 real time Performance Monitor.

HARDWARE Requirements

1. Record and Time Stamp Selective 4700 Processing Events. This includes Engine Priority Level Changes, going into WAIT STATE, coming out of WAIT STATE, and capturing specified address references.
2. The Time Stamps will be in 1 US increments.
3. There will 3 physical entities interconnected by cables:
 - A REDRIVER CARD which will mount directly on the top connectors of the 4700 Engine card.
 - A LOGIC BOX having a self contained 110 VAC Power Supply.
 - An IBM PS2 Compatible MICRO CHANNEL (U-Channel) ADAPTER CARD.

SOFTWARE Requirements

There is at least two major pieces and the potential for a third major piece. These entities are:

- Measurement Setup and Control Program
- A Post Measurement Reduction and Converter Program
- An optional PC Data Reduction Program

Measurement Setup and Control Program

A program that uses some form of Presentation Manager to interface with an Operator and communicates with the LOGIC BOX using Device Drivers provided by the Original Manufacturer of the PS2 U-Channel Adapter Card.

1. Main Control Panel

The main control panel will have a Field for 8 character File Name, a 60 character test description field, and 16 individual fields for address comparers; i.e. four characters for each address comparer, however, the actual hardware (LOGIC BOX) may have no or only 4 address comparer circuits.

The main panel structure will be such that any field can be changed or modified up till when a commitment is made to start a Measurement Run.

The main control panel will outline the control keys required to initiate and stop a Measurement Run.

2. Key Program Initialization Functions

- RESET all key and critical LOGIC BOX Control Circuits, prior to initializing a run.
- Check the LOGIC BOX STATUS Byte(s) to make certain the BOX is error free prior to START.
- Check for the available of space in the interim buffer area. An Operator settable incremental value should be available on a second PANEL. If incremental space value is unavailable, inform the Operator for corrective action.

- Check that the File Name assignment is not a duplicate assignment of an existing Measurement Data Set. If a conflict exists, give the Operator a window to choose whether to rename the file to be made, or to Delete the existing file.
- OPEN the Measurement File; i.e. if using a VDISK or Hard Disk for the interim storage of the current Measurement Run.
- As part of the START Operation and prior to the actual START Command to the LOGIC BOX, the program will store the Comment Field plus other miscellaneous data in the first 256 bytes of the interim Data Set; i.e. lowest addressed positions.
- START the LOGIC BOX; i.e. begin a measurement run.
- Give the Operator a panel indicator (like an incrementing count) or a noise (Speaker) indication that the PC is receiving Measurement Records.
- Handle the LOGIC BOX requests for READ OUTS expediently.
- Store all Measurement Records sequentially, in the same order as they are received for the PC.
- Have logic checks for internal PC Overruns.
- It is preferable that each LOGIC BOX READ OUT include a STATUS READ, to see if there has been an Overrun.
- Assuming Overrun Checks, we need to assign a Special Record ID for overruns and the program should store this Record with the current block of data being placed in interim storage.
- STOP the LOGIC BOX; i.e. Measurement Run Complete.
- Flush any untransferred data from the LOGIC BOX.
- Call the Post Reduction and Converter Program upon completion of the Measurement Run.

3. UTILITY PANEL Support

Setup and Control Program should have an optional panel, that would enable the Operator to perform basic checks and exercising of the LOGIC BOX Interface. The main purpose would be to facilitate troubleshooting.

Following are some examples of these Functions:

- READ OUT the LOGIC BOX STATUS Register, and present its contents; i.e. preferable in its raw hexadecimal form and in an interpreted form.
- Issue a standalone LOGIC BOX RESET Signal or optionally a series of RESET Signals; e.g. operator having a choice of nn RESETS that will be issued at a nn US Rate.
- Issue a standalone LOGIC BOX START Signal or optionally a series of START Signals; e.g. operator having a choice of nn RESETS that will be issued at a nn US Rate.
- Issue a standalone LOGIC BOX STOP Signal or optionally a series of STOP Signals; e.g. operator having a choice of nn RESETS that will be issued at a nn US Rate.

Post Measurement Reduction and Converter Program

Data must be moved from the interim buffer, converted to be compatible with the existing UCAM Records, and place on Hard DASD FILE.

Following is detail on Compatibility Requirements:

- All Final Measurement Records will be three bytes long.
The low addressed byte indicates record type; details will be covered separately.
The middle byte contains a TIMER VALUE of 0 thru 256; i.e. 0h thru FFh.
The high addressed byte is used for more detail on the Measurement Record.
- All Measurement Records in the Final Data Set will be in the proper sequential order. The first Measurement Record will in the lowest addressed position for data, the last Measurement Record will be in the highest address position, and all interim Measurement Records will also be so ordered.
- Special TIMER Processing will be required on a large number of Measurement Records, regardless of the exact Hardware Timing Scheme chosen.

This required TIMER Compatibility Processing may result in the consolidation of some Records (if 1 Byte Timer is used), in the creation of extra TIMER Records (if the TIMER >1 Bytes) and in the adjustment of a many (probably 20 % to 67 %) TIMER VALUES in NON-TIMER Measurement Records (if the TIMER is >1 Byte).

This programming overhead (coding and processing time) is considered incidental to having an optimum Hardware Design; i.e. a design that is rock solid and most likely to produce MEASUREMENT RECORDS in their exact order of occurrence, regardless of recording priorities or buffer control problems.

That said, we will expect that TIMER Processing will be handled by a tightly coded and highly efficient routine.

Optional PC Data Reduction Program

Presently all final Data Reductions are made on an IBM HOST SYSTEM; i.e. 370VM System, using a PLI Program. This program has a number of optional Output Formats. All have a Summary of the Measurement Run; e.g. number of entries on each Engine Priority Level, total time on each priority level, total processing time, total WAIT Time, etc.

It would be ideal to have these reduction capabilities in the Local PC controlling the Measurement Run.

Hardware Details

LOGIC BOX

1. The LOGIC BOX should be designed to be portable. It is desirable for this box to have handles, internal storage space for all cabling and to weigh less than 20 pounds.
2. It is also desirable to have a carrying box for the Redrive Card and the U-Channel Adapter Card. This could be in the Logic Box or be separate.
3. All LOGIC BOX Logic Circuits should be readily accessible for scoping. This can be accomplished by fold-outs, pop-ups, extenders, or having all logic on one accessible card.

Note: If extenders are used, it assumed that Standard OEM Proto-typing Boards are used and that time critical circuitry will not bridges between cards; e.g. circuitries that would be sensitive to +/- 16 NS timing variations.

4. The initial design of the logic cards should NOT commit more than 70 % of the real estate and the unused space should be distributed; i.e. space for corrections and minor expansion of function.
5. The 1 US TIMER will be driven by a local oscillator.
6. The TIMER value will be recorded only when it is in Study State; e.g. 750 NS per 1000 NS.
7. The recorded TIMER VALUE will be 1, 2 or more bytes; i.e. atleast 8 bits, or 16 bits, or ???
8. All TIMER OVERFLOWS will be recorded.
9. The TIMER can be reset to zero on each recorded event or it can be free running. The latter is probably more reliable and less complex.

Note: Regardless of the exact design chosen for the TIMER, the PC's POST PROCESSING will have to adjust, rehandle, calculate overflows, and possible remap all timer entries.

10. If reset on event timing is used, all event records MUST BE sequentially recorded in the order of occurrence.
11. The LOGIC BOX will need priority circuitry to prevent multiple or overlaying records; i.e. a TIMER overflow concurrent with a Engine Priority Change, or an Address Hit concurrent with a Priority Change, etc. Whatever scheme is chosen should be reviewed with Charlotte.
12. A Event Record will consist of 1 byte (8 bits) indicating the event type, 1 or more bytes for the TIMER, and one byte for detail. Therefore each Event Record will be at least 3 bytes in length.
13. The LOGIC BOX should be able to make valid records at a 1 US rate, for up to 16 events; i.e. 16 events, in 16 USs.
14. The LOGIC BOX should be able to sustain an operation having an average of 1 Recorded Event, every 32 US.

Assuming a 3 Byte Event Record, this would be an effective recording rate of 94 KBytes/sec., 125 KBytes/sec. for 4 bytes, etc.

15. The LOGIC BOX must be to sustain the above rates, while uploading the PC with previously captured Event Records.

The allusion here is atleast a PING-PONG Buffering Scheme. One buffer is active with recording, while the other is being emptied; i.e. uploaded to the PC.

16. The LOGIC BOX will asynchronously signal the PC when it has data to upload.
17. The LOGIC BOX must have sufficient Event Record Buffers to maintain the stated recording rates, while waiting for the PC to digest previously uploaded data; i.e. maximum PC Interrupt recognition time, the upload time of the incremental buffer, the application processing time, data move time (e.g. from PC I/O Buffering to a VDISK), etc.
18. The LOGIC BOX must have circuitry to intercept Asynchronous Interrupts from the PC; e.g. to START RECORDING, set Addressing Traps, STOP RECORDING, etc.

19. The LOGIC BOX should have a STATUS REGISTER to indicate when it has detected an internal error (e.g. buffer overruns, priority lock outs, stop state, run state, etc.)
20. The LOGIC BOX's STATUS REGISTER should be readable by the PC on demand.
21. The LOGIC BOX will have logic to complement the PS2 U-Channel Adapter Card. Namely it will have logic for the Protocol Standard Chosen and it will support two way asynchronous communications.

PS2 U-Channel Adapter Card

1. It is anticipated that the PS2 U-Channel Adapter Card will be stock OEM or IBM Card. The Communication Protocol Chosen should support at least 4 times the desired average transfer rates; e.g. 125 KBytes times (X) 4 = .5 MBytes/sec. The Protocol must all so support asynchronous traffic in either direction.

Two potential candidates are:

- IEEE 422

The contracting Charlotte Group has a good experience base on this item.

- IEEE 488

Professor Krilly has experience on this protocol.

2. It is assumed that the original manufacturer of the Adapter Card will provide the necessary PC Device Driver Code.
3. The PC Adapter will adhere to the standard IBM U-Channel Adapter Card Form Factor, i.e. it will fit in a IBM PS2P70's long slot, with standard (over the counter) I/O Connector and cabling.

REDRIVER CARD Detail

1. This card will be approximately 3 by 7 inches, with connectors on the bottom, to interface with 4700 Engine Card and connectors at the top, to interface with the LOGIC BOX.
2. The prime purpose of this card is to be a passive redriver; i.e. to redrive the engine lines to the LOGIC BOX.
3. The REDRIVER CARD power source should be separate for the data transmission cable and be optional sourced.

From a ground loop noise stand-point it is desirable that the REDRIVER CARD be powered by the 4700 Engine Supply. However it is not easy to tap this source, so we should have an alternate source, in the LOGIC BOX.

Cabling Detail

1. Standard Cables (over the counter) should be used where ever possible.
2. The Signal Cable(s) from the REDRIVER CARD to the LOGIC BOX should be around 8 feet long.
3. The Power Cables for the REDRIVER CARD (+5 VDC) and for the LOGIC BOX (120 VAC) should also be around 8 feet.
4. The Signal Cable(s) from the LOGIC BOX to the PS2 U-Channel Adapter Card should optionally be 10 or 25 feet. Portable storage provisions need only accommodate 10 feet.

Software Details

Measurement Records - The Type, Byte 1

"00h" : A basic TIMER Record (raw of the LOGIC BOX).

Byte 2 of the Measurement Record should be "FEh" even though the real hardware TIMER cause this RECORDING will probably be set to "00h" or "FFh."

Byte 3 of the Measurement Record should be set to "00h".

"02h" : An Engine Level Change RECORDING. It is also used to indicate that the Processor has left the WAIT STATE; i.e. is now actively Processing Code.

Byte 2 of the Measurement Record should indicate the number of hexadecimal "US"s that have elapsed since the preceding Measurement Record. This is based on the design of our present Measurement Tool ("UCAM"), which resets the Hardware TIMER on every RECORDING.

Byte 3 of the Measurement Record is divided into 2 nipples of 4 bits each. The first nipple is the hexadecimal representation of the new level. The second nipple is the old engine priority level.

- The levels range from "00h" (Level 0) to "07h" (Level 7).
- The Levels are determined by partially capturing the RAM Addressing being used to store the Processor's Local Registers (for the old level) and doing the same for the retrieve of the Processor's new Local Registers (for the new level). These numbers need to be buffered in a "Dedicated Hardware Register" for two reasons. One is, there is a time delta between these two RAM References, so a MEASUREMENT RECORD can not be made until after the last data capture; i.e. the new Level number. The second reason is these numbers need to be recalled, when the Processor comes out of the WAIT STATE.

"04h" : Processor is going in to the WAIT STATE; i.e. engine is idle.

Byte 2 of the Measurement Record should indicate the number of hexadecimal "US"s that have elapsed since the preceding Measurement Record. This is based on the design of our present Measurement Tool ("UCAM"), which resets the Hardware TIMER on every RECORDING.

Byte 3 of the Measurement Record should be set to "00h".

"09h" : An address compare hit; i.e. LOGIC BOX detected a predefined address being referenced by the 4700 Processor.

Byte 2 of the Measurement Record should indicate the number of hexadecimal "US"s that have elapsed since the preceding Measurement Record. This is based on the design of our present Measurement Tool ("UCAM"), which resets the Hardware TIMER on every RECORDING.

Byte 3 uses individual BITS to indicate which Comparer out of 8 possible in each set of comparers.

"0Ah" : An address compare hit; i.e. LOGIC BOX detected a predefined address being referenced by the 4700 Processor.

Byte 2 of the Measurement Record should indicate the number of

hexadecimal "US"s that have elapsed since the preceding Measurement Record. This is based on the design of our present Measurement Tool ("UCAM"), which resets the Hardware TIMER on every RECORDING.

Byte 3 uses individual BITS to indicate which Comparer out of 8 possible in each set of comparers.

"60h" : A PACKED TIMER Measurement Record. This will produced by the "Post Measurement Reduction and Converter Program."

Byte 2 of the Measurement Record should be "FEh" even though the real hardware TIMER cause this RECORDING will probably be set to "00h" or "FFh."

Byte 3 will the hexadecimal representation of the number of 256 US Time Periods represented by this Measurement Record.

Equipment to be Loaned

1. "HP" 1630G Data Analyzer
2. An "IBM" PS/2 having an "INTEL" 80286 Engine and a Micro-Channel.

Documentation Requirements

1. 2 sets of Logic Diagrams, one being "MASTERS".
2. 2 sets of Wiring Charts of all decrete wiring.
3. 2 sets of any LAN CARD Layouts.
4. 2 sets of an Operations Manual

APPENDIX B

TIMER FIELD LENGTH DERIVATION

This section describes work done to derive the optimal length of the Timer Field of the Event Record. This work is done to reduce the amount of data storage and data transmission between the monitor and the host PS/2, and these are directly affected by the length of the Event Record. The Event Record consists of three fields of information, the Command field, the Timer field, and the Comment field. The Command and Comment fields have already been specified as 1 byte each, leaving only the Timer field length undefined.

The length of the Timer field affects the data storage in two ways. Naturally, the greater the length of the timer field, the greater the storage and transmission bandwidth needed for Event Records, and this criteria would indicate as small a Timer field as possible. On the other hand, a decrease in the Timer field causes an increase in the number of Timer Overflow Events that occur, which take storage and transmission space. This criteria would indicate as large a Timer field width as possible. Below the Event rate and the Timer Overflow rate are used to derive the optimal length of the Timer Field.

The byte rate is a function of two separate but related rates, the Event rate and the Timer Overflow rate, shown in Equation 1. The Event rate, specified by IBM, is an inherent data rate, because this rate covers Events on the 4702, and is beyond design control. The Timer Overflow rate is overhead and is design controllable through manipulation of the Time field length. Both the Event rate and the Timer Overflow rate, with units of Events/second, are multiplied in equation 1 by the record length, with units of bytes/Event, to form the Byte rate.

$$\text{Byte Rate} = (\text{Event Rate} + \text{Timer Overflow Rate})(\text{Record Length}) \quad (1)$$

The Timer Overflow rate is a function of the length of the Time field, but not in a linear relationship. Timer Overflow Events occur when the timer value changes from the highest value to 0, and the highest value will vary as a power of 2 with the length of the timer. The actual equation for the Timer Overflow rate is given in Equation 2, assuming a 1 MHz clock. The Timer Overflow rate given in equation 2 again has units of Events/second.

$$\text{Timer Overflow Rate} = \frac{10^6}{2^{(\text{Number of Timer Bits})}} \quad (2)$$

The number of Timer bits can be expressed as a function of the Time field length to facilitate comparison. With this change, equation 2 becomes equation 3.

$$\text{Timer Overflow Rate} = \frac{10^6}{2^{(8(\text{Time Field Length}))}} \quad (3)$$

An equation for the Timer Overflow rate as a function of the Time field length has been determined. The length for the Event Records, or the Record Length, is also a function of the Time field length, and it equal to the Time field length plus two (to account for the Command and Comment bytes). By expanding equation 1 and substituting equation 3 for the Timer Overflow rate, equation 4 is resulted.

$$\begin{aligned} \text{Byte Rate} = & (\text{Event Rate})(\text{Time Field Length} + 2) \\ & + \frac{10^6}{2^{(8(\text{Time Field Length}))}} (\text{Time Field Length} + 2) \quad (4) \end{aligned}$$

Examining equation 4 two variables exist, Event Rate and Time field length. To simplify equation 4 and to facilitate comparison, the Time field length will be held constant and the Event Rate will be varied. By holding the Time field length constant, the section of equation 4 dealing with the Timer Overflows becomes a constant, and equation 4 describes a linear

relationship between the Event Rate and the Byte Rate. The Timer Overflows in effect describe an intercept for the linear equation and the Time Field, with the addition of two for the Command and Comment bytes, form the slope of the line.

The purpose for the derivation of equation 4 is the determination of how to reduce the byte rate through the variation of the Record Length. The lengths under consideration are discrete values of 1, 2, and 3 bytes. The specification for the Event rate is set as a maximum of 31250 Events/second and a minimum of 0 Events/second, yielding the domain for the Event rate. The Time field length will be held constant for each of the three values of 1, 2, and 3 bytes. Table B-1, Graph B-1, and Graph B-2 show the values of the byte rate vs. the Event Rate, as given in equation 4, for the three values of Time Length.

Table B-1 and Graph B-1 show the entire range for the Event Rate, from 0 to 31250 Events/second. Analyzing Graph B-1 and Table B-1, it is becomes apparent that the byte rate for the 3 byte Time field is larger than the byte rate of the 2 byte Time field except very near 0 Events/second, and therefore offers no advantages. The 24 bit Time field length will be eliminated from consideration. Analysis of the 8 and 16 bit Time field is more complicated. Looking at Table B-2 and Graph B-1, the byte rate for the 2 byte Time field is smaller than the 1 byte field until the Event rate is 11658 Events/second or greater. At this point, the 8 bit Time field offers a lesser byte rate. This change is due to the fact that more Events are

Table B-1: Byte Rate Vs. Event Rate.

<u>Event Rate</u>	<u>1 Byte Timer Field</u>	<u>2 Byte Timer Field</u>	<u>3 Byte Timer Field</u>
0	11719	61	0
50	11869	261	250
61	11902	305	305
100	12019	461	500
1000	14719	4061	5000
5000	26719	20061	25000
5650	28669	22661	28250
5859	29296	23497	29295
5890	29389	23621	29450
6000	29719	24061	30000
11000	44719	44061	55000
11500	46219	46061	57500
11658	46693	46693	58290
11750	46969	47061	58750
12000	47719	48061	60000
16000	59719	64061	80000
17000	62719	68061	85000
23000	80719	92061	115000
29000	98719	116061	145000
30000	101719	120061	150000
31000	104719	124061	155000
31250	105469	125061	156250

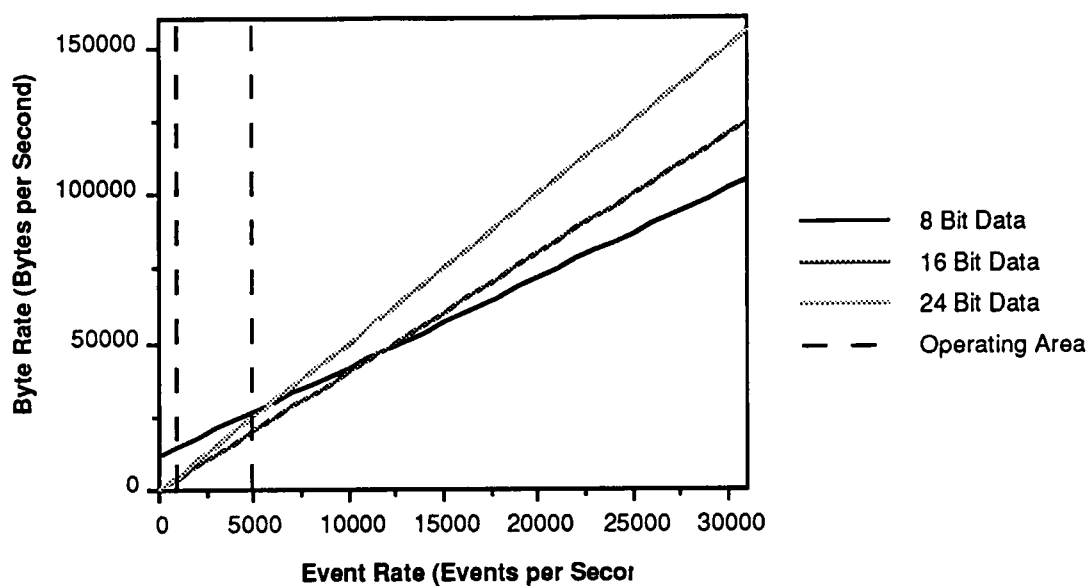


Figure B-1: Graphical results of equation 4, with operating area shown.

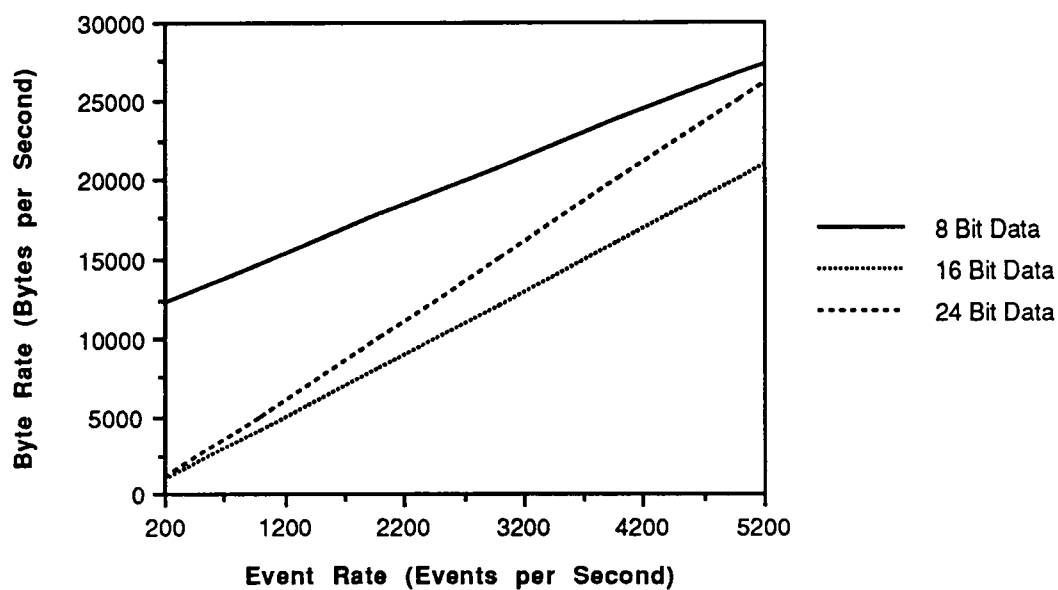


Figure B-2: Byte Rate v. Event Rate with Emphasis on the Operating Area.

occurring between the Timer Overflows, reducing the overhead effect of Timer Overflows. As the effects of Timer Overflows are lessened, the effect of the larger record length, due to an increase in the Time Field length are more readily shown. After further consultation with IBM on the 4702 operations, it was determined that the Event rate centered mostly in the 200 to 5000 Events/second domain, and that the maximum value of 31250 Events/second was hardly ever reached. The area of the Event rate range from 200 to 5000 Events/second is called the operating range, because most operations on the 4702 will occur within these Event rates. Graph B-2 shows an expansion of the Curves in the operating range. Analysis of these curves indicates that the byte rate for the 2 byte Time field is less than the 1 byte Time field length, due to the increased overhead effect of Timer Overflows.

From this analysis, it was determined that a Time field length of 2 bytes was optimal. The Event Record will therefore consist of the Command field, the Timer field, and the Comment field, with 1, 2, and 1 byte respectively. The Event Record will therefore be 4 bytes in length. The Timer field for the Event Record is different from the original as specified by IBM, as the Timer field length is one byte longer, and the timer used in non-resetting. This change will have to be accounted for in the PS/2 post processing and conversion program, but was deemed worthy due to the ease of timer construction and the reduced .

APPENDIX C

LBI: PS/2 CONTROL SOFTWARE LISTING AND LBF: MONITOR FIRMWARE LISTING

This appendix gives a listing of the PS/2 software and Monitor firmware used to communicate with and control the Monitor hardware. The PS/2 software is written in the C programming language and was compiled using the Turbo C compiler. Two files are shown here; the first is the header file, LBIV207.H, and the second is LBIV207.C, the main body of code. LBI (Logic Box Interface) is the program name, and V207 is the version number, 2.07.

The Monitor firmware is written in a combination of FORTH and Assembly language for the 68CH11 microprocessor and uses the embedded FORTH Kernel from New Micros, Inc. of Dallas, Texas.

LBIV207.H

```

/*****

lbiv210.h - Header file for the Logic Box Interface Code

This file contains most of the structure and definitions
for the Logic Box Interface Code.

*****/

/* ----- General Definitions ----- */

typedef unsigned int      uint;
typedef unsigned char     uchar;
typedef unsigned short    ushort;
typedef unsigned long     ulong;

/* ----- Include Files ----- */

#include <stdio.h>          /* needed for print operations      */
#include <io.h>             /* needed for low-level file handling */
#include <fcntl.h>          /* needed for low-level file handling */
#include <sys/types.h>
#include <sys/stat.h>
#include <conio.h>          /* needed for low level console fnc  */
#include <dos.h>            /* needed for getting file info      */
#include <stdlib.h>         /* needed for some char conversion   */
#include <ctype.h>          /* needed for some char conversion   */
#include <errno.h>         /* list of error constants           */

/* ----- General Definitions ----- */

#define CRASHTHRESHOLD    2000 /* test constant for Hardware crash test */
#define DMA_FLAG          1   /* 1 = use DMA, 0 = No DMA                */
#define ID_SCSI           0   /* SCSI ID for the PS/2                  */
#define CMD_SIZE          0x02 /* Size of CMD Blocks from Monitor       */
#define OUT_BUFF_SIZE     2   /* size of the SCSI output buffer        */
#define DATA_LENGTH      512 /* Length of Transfers from Monitor      */
#define REFRESHSTATUS_VALUE 1000 /* how many loops befor refreshstatus */

/* ----- Defn. for Command bytes ----- */

#define SET_CONTROL       0x00 /* Command for Setting Control Values    */
#define RESET_MONITOR     0x01 /* Command for Resetting Monitor         */
#define START_MONITOR     0x02 /* Command for Starting Monitor Ops      */
#define STOP_MONITOR      0x03 /* Command for Stopping Monitor Ops     */
#define SEND_STATUS       0x04 /* Command for Get Status of Monitor     */
#define ACK_MONITOR       0x05 /* Command to Check Monitor connect     */

/* ----- CSM Structures ----- */

typedef struct Target_Table_Entry { /* Structure used for Target */
    uchar      CMD_Code;
    uchar      CMD_Size;
    void      (*far *Funct_Pntr)();
} Target_Table_Entry;

```

```
typedef struct Init_Table {                                /* Struct used for Init. SCSI */
    uchar                SCSI_ADDR;
    Target_Table_Entry   far *tbl_addr;
    uchar                rsvd[8];
    int                  base_add;
    uchar                dma_chan;
    uchar                int_level;
}Init_Table;

typedef struct CMD_Block {                                /* Struct used for Sending CMDs */
    uchar                OP_Code;
    uchar                LUN;
    uchar                MSB;
    uchar                LBA;
    uchar                LSB;
    uchar                Control;
}CMD_Block;

struct OEM_Table {                                        /* Struct for entire SCSI send */
    CMD_Block far        *CMDBlk_addr;
    uchar far           *DATABlk_addr;
    ulong               Expect_Length;
    ulong               Actual_Length;
    uint                Timeout;
    uint                Blk_A;
    uint                Blk_B;
    uchar               Status;
    uchar               MSG;
    uchar               Control_Lines;
    uchar               SCSI_ID;
    uchar               Parity;
    uint                Error_Code;
}OEM_Table;

/* ----- CSM SCSI Function Definitions ----- */

void send_data(uchar far *buffer, ulong size, int dma);    /* SCSI send */
void recv_data(uchar far *buffer, ulong size, int dma);    /* SCSI receive */
void send_msg(uchar msg);
void send_stat(uchar stat);
void bus_free(void);                                     /* frees bus */

/* ----- Structures for LBI ----- */

typedef struct StatusTbl {                                /* Struct holding status flags */
    int                Acked;                               /* flag for Monitor connection */
    int                Cntrl_Val;                           /* holds current Mon CNTR VAL */
    int                Mongo;                               /* flag for Monitor Started */
    int                Monstop;                             /* flag for Monitor Stopped */
    int                MonStat;                             /* Var. for Last Mon. Status */
    int                Reception;                           /* flag for Receiving data */
    int                SCSI_Error;                          /* flag for SCSI error occurred */
    int                SCSI_Overflow;                       /* flag for scsi buffer overflw */
    int                SCSI1_Full;                          /* flag for scsi buffer 1 full */
    int                SCSI2_Full;                          /* flag for scsi buffer 2 full */
    int                File_Error;                         /* error writing file */
}
```

```
int      Last_Usr_CMD;      /* last keyboard Cmd from user */
long     bytecount;        /* number of bytes received */
}StatusTbl;
```

LBIV210.C

/*****

LBI - Logic Box Interface version 2.10

This code will form the Logic Box Interface. This code will control and respond to actions of the Monitor.

This code will control the actions of the monitor by issuing five commands to the monitor. The commands are:

Set_Control	- Sets capture of level changes and waits.
Reset_Monitor	- Resets all Monitor functions and actions.
Start_Monitor	- Starts Monitor capture of 4700 data.
Stop_Monitor	- Stops Monitor capture of 4700 data.
Send_Status	- Retrieves the Monitor Status.

The Logic Box Interface will respond to two commands from the monitor. These commands are:

Receive_Data	- Retrieve data from Monitor to RAM disk.
Ack_Connection	- Assure connection with the PS/2.

One header file is used with this program. The file is

lbiv207.h

This version of code is different from version 2.00 because it uses a SCSI Buffer area on the PS/2 instead for having the interrupt routine write the information to file. The main loop is now responsible for seeing the data, by flags, and writing the information to file.

This version is different from version 2.02 because I have added a status structure called StatusTbl which keeps up with the status of the System(both monitor and ps/2). This information is also printed to the screen.

This version is different from version 2.03 because I have tried to update the Status printing functions and make any bug fixes that are necessary. The updatestatus() function is changed, as well as others.

In version 2.07, I have changed the options so that I can redraw the screen if needed.

In version 2.09, I have added the header information to the output files so that the files may retain compatibility with the older IBM UCAM format.

For version 2.10, I have made some small fixes and gotten the code ready for the final version.

*****/

```
#include "lbiv210.h"          /* header for this version of lbi    */
```

```
/* ----- Global Function Prototypes ----- */
```

```
void far Target01_Routine(char *cmdbytes, char *msgbyte, int Init_ID);
void far Target00_Routine(char *cmdbytes, char *msgbyte, int Init_ID);
uchar Check_Disk_Space(int Disk_Name);
```

```
/* ----- Global Structures ----- */
```

```
struct OEM_Table OEM_Tbl;          /* CSM Table for sending SCSI Data */
struct CMD_Block Command;          /* CSM Table for holding command  */
```

```
struct StatusTbl OldStatTbl={1,0,1,1,0,0,0,0,0,0,0,0,(long)0}; /* Global Stat Tables */
struct StatusTbl NewStatTbl={0,0,0,0,0,0,0,0,0,0,0,0,(long)0};
```

```
/* ----- Global Variables ----- */
```

```
char SCSI_Buffer1[DATA_LENGTH];    /* SCSI input data buffer        */
char SCSI_Buffer2[DATA_LENGTH];
char fullheader[256];              /* header array */
```

```
int Oldest_SCSI_Buffer = 1;        /* Indicates which SCSI buffer is newer */
```

```
char Out_Buff[OUT_BUFF_SIZE];
char fname[64];                    /* filename char array            */
int Data_FH = -1;                  /* File Handle, SCSI Data        */
int Drive_Name=3;
int barcount = 0;
int updatefname = 0;               /* status update flag */
int crashtest = 1;
```

```
/* ----- Main Program Entry ----- */
```

```
main()
```

```
{
```

```
    int key, Write_Stat, i, crashcount = 0, crashtesten = 0, test, refreshcount = 0;
    long Stat;                      /* long for status returned from Logic Box */
    char strtemp[33];               /* defines temporary string array */
    char *strpntr;                  /* defines pointer returned by ultoa */
```

```

/* ----- Main Prototypes and Structures ----- */

void printhead(void); /* prints command sel hdr */
void screenmsg(char * string, char *cntstring, int param_in); /* prints message */
*/
void refreshstatus(void); /* updates screen status printout */

int getyesno(void); /* gets yes, no answer from user */

char * getheader(void); /* subroutine to get and display the header */

int SetCntrMon(int CV); /* func to set Mon Cntrl */
int StartMon(void); /* Starts Monitor, returns error */
int StopMon(void); /* Stops Monitor, returns error */
int ResetMon(void); /* Resets Monitor, returns error */
int AckMon(void); /* Checks Monitor Connection */
long RetrieveStatus(void); /* Returns Status and Error code */

void InitOEMTable(void);

struct Target_Table_Entry Command_00; /* sett structs for */
struct Target_Table_Entry Command_01; /* the Target Table */
struct Target_Table_Entry Command_NULL;
static struct Target_Table_Entry Target_Table[3]; /* Target Table Array */
struct Init_Table UTUCAM_Init_Table; /* CSM Init Table */

/* ----- Target Table Entry Initialization ----- */

Command_00.CMD_Code = 0x00; /* load command byte for */
Command_00.CMD_Size = CMD_SIZE; /* 00 Target Table */
Command_00.Funct_Pntr = Target00_Routine;

Command_01.CMD_Code = 0x01; /* load command byte for */
Command_01.CMD_Size = CMD_SIZE; /* 01 Target Table Entry */
Command_01.Funct_Pntr = Target01_Routine;

Command_NULL.CMD_Code = NULL; /* loading NULL */
Command_NULL.CMD_Size = NULL; /* Target Table Entry */

/* ----- Full Target Table Initializaiton ----- */

Target_Table[0] = Command_00; /* command byte 00 entry */
Target_Table[1] = Command_01; /* command byte 01 entry */
Target_Table[2] = Command_NULL; /* must end Target_Table */

/* ----- Initializing the CSM board ----- */

UTUCAM_Init_Table.SCSI_ADDR = ID_SCSI; /* sets up init struct */
UTUCAM_Init_Table.tbl_addr = Target_Table;
UTUCAM_Init_Table.base_add = -1; /* not used in CSM-105 */
UTUCAM_Init_Table.dma_chan = -1; /* not used in CSM-105 */
UTUCAM_Init_Table.int_level = -1; /* not used in CSM-105 */

CSMINIT(&UTUCAM_Init_Table); /* Init Target Mode */
InitOEMTable(); /* Init Initiator Mode */

```



```

/* ----- Initializing the SCSI buffer area for testing -----*/

for (i = 0; i < 512; i++)
    SCSI_Buffer1[i] = 0xBC;

/* ----- Monitor Control Code -----*/

key = '5'; /* sets up to check for monitor status when starting */

printf("\033[2J");          /* clears the screen */
printhead();               /* Prints out user header */
refreshstatus();

do /* beginning of the loop */
{
    refreshcount++;
    if(refreshcount ==1000)
    {
        refreshstatus();
        refreshcount = 0;
    }

    if(key != 0)
    {
        switch(key)
        {

/*****
        case '1' - Overall Function: Setting Control Register on Monitor.
            - Actions: Asks user whether to capture levels and waits.
                        Forms a control value to send to Monitor.
                        Sends Monitor information accross SCSI.
                        Performs Error Checking afterwards.
*****/

        case ('1'): printf(" 1");

            screenmsg("Should Waits be Captured?", "", 0);

            NewStatTbl.Cntrl_Val = getyesno(); /* 1=yes 0=no*/

            screenmsg("Should Level Changes be Captured?", "", 0);

            NewStatTbl.Cntrl_Val = (NewStatTbl.Cntrl_Val | (getyesno()*2));

            if (SetCntrMon(NewStatTbl.Cntrl_Val))
            {
                screenmsg("A SCSI error occurred!", "\a", 0);
                screenmsg("The error was %i", "", OEM_Tbl.Error_Code );
                NewStatTbl.SCSI_Error=OEM_Tbl.Error_Code; /* update status tbl */
            }
            else
            {
                screenmsg("Monitor Control was set.", "", 0);
                NewStatTbl.SCSI_Error=0; /* reset SCSI_Error in table */
                NewStatTbl.Last_Usr_CMD = 1;
            }
        }
    }
}

```

```

        NewStatTbl.Acked=1;
    }
    break;

/*****
case '2' - Overall Function: Starting Monitor Capture of Events.
    - Actions: Attempts to Start Monitor.
               Informs users whether an error occurred.
*****/

case ('2'): printf(" 2");

    screenmsg("Store data to what filename? ", "", 0);
    gets(fname);

    getheader();          /* set up the header values */

    if (Data_FH != -1)
        close(Data_FH);

    Data_FH = open(fname, O_WRONLY | O_CREAT | O_BINARY,
                   S_IREAD | S_IWRITE);

    updatefname = 1; /* status flag so refreshstatus will update */

    if(Data_FH == -1)
    {
        screenmsg("Data File could not be opened!!!!", "\a", 0);
        NewStatTbl.File_Error=1;      /* indicate file error */
        break;
    }

    Write_Stat = write(Data_FH, fullheader, 256); /* writes header */

    if(Write_Stat != 256)
    {
        screenmsg("Error writing header!!!", "\a", 0);
        screenmsg("Monitor not started.", "", 0);
    }
    else
    {
        NewStatTbl.Mongo = 1;
        NewStatTbl.Monstop = 0;
        NewStatTbl.bytecount = 0;

        if (StartMon())
        {
            screenmsg("Monitor not Started - SCSI error!", "\a", 0);
            NewStatTbl.SCSI_Error=OEM_Tbl.Error_Code;
        }
        else
        {
            screenmsg("Monitor Started!", "", 0);
            NewStatTbl.Mongo=1;      /* indicate Monitor Started */
            NewStatTbl.Monstop=0;
            NewStatTbl.Last_Usr_CMD=2;
            NewStatTbl.Acked=1;

```

```

        NewStatTbl.bytecount = 0; /* reset the byte count */
    }
}
break;

/*****
case '3' - Overall Function: Stopping capture of Monitor Events.
- Actions: Attempts to Stop Monitor.
          Informs users whether an error occurred.
*****/

case ('3'): printf(" 3");

    if (StopMon())
    {
        screenmsg("Monitor not stopped - SCSI error!", "\a", 0);
        NewStatTbl.SCSI_Error=OEM_Tbl.Error_Code;
        NewStatTbl.Monstop = 1;
    }
    else
    {
        screenmsg("Monitor Stopped.", "", 0);
        NewStatTbl.Monstop=1; /* indicate Monitor stopped */
        NewStatTbl.Last_Usr_CMD=3;
        NewStatTbl.Acked=1;
        NewStatTbl.Reception = !OldStatTbl.Reception;
        barcount = -1;
        NewStatTbl.SCSI_Overflow = 0;
    }
    break;

/*****
case '4' - Overall Function: Resetting Monitor.
- Actions: Attempts to Reset Monitor.
          Informs users whether an error occurred.

- Note: This does NOT use the SCSI bus Reset.
        This uses a reset command.
*****/

case ('4'): printf(" 4");

    if (ResetMon())
    {
        screenmsg("Monitor not reset - SCSI error!", "\a", 0);
        NewStatTbl.SCSI_Error=OEM_Tbl.Error_Code;
    }
    else
    {
        screenmsg("Monitor Reset.", "", 0);
        NewStatTbl.Mongo=0; /* update status table */
        NewStatTbl.Monstop=0;
        NewStatTbl.Last_Usr_CMD=4;
        NewStatTbl.Acked=1;
        NewStatTbl.bytecount = 0; /* resets the byte count */
        barcount = -1; /* resets the bar count */
    }

```

```

    }
    break;

/*****
case '5' - Overall Function: Retrieves Monitor Status.
    - Actions: Attempts to get Monitor status.
                Informs users whether an error occurred.
                Interpretes the Status byte for user.
*****/

case ('5'): printf(" 5");

    Stat = RetrieveStatus();
    if (Stat & 0xFFFF0000)
    {
        screenmsg("SCSI error retrieving status!", "\a", 0);
        NewStatTbl.SCSI_Error = OEM_Tbl.Error_Code;
    }
    else
    {
        screenmsg("The Status byte is %X.", "", Stat);
        NewStatTbl.MonStat = (int)Stat;
        NewStatTbl.Acked=1;
        NewStatTbl.Last_Usr_CMD = 6;
        NewStatTbl.Cntrl_Val = 3 & (NewStatTbl.MonStat >> 3);
    }
    break;

/*****
case '6' - Overall Function: Redraw the screen.
*****/

case ('6'): printf("\033[2J"); /* clears the screen */
    printhead();
    printf(" 6");
    screenmsg("Screen Redrawn.", "", 0);
    break;

/*****
case '7' - Overall Function: Exit to the Operating System.
    - Actions: Exits.
*****/

case ('7'): close(Data_FH); /* closes data file */
    UNHOOK(); /* Unhooks SCSI Int. */
    printf("\033[2J"); /* clears the screen */
    printf("Goodbye!");
    exit(1); /* Exits to DOS */

/* Test code for seeing if the hardware has crashed */

case ('8'):
    if (crashtesten)
    {
        crashtesten = 0;
        screenmsg("Crash test disabled.", "", 0);
    }

```

```

    }
    else
    {
        crashtesten = 1;
        screenmsg("Crash test enabled.", "", 0);
    }
    break;
}
}

if (kbhit()) /* checks whether keyboard has been hit */
{
    key=getch(); /* gets key for next time through loop */
    if (NewStatTbl.Mongo && !NewStatTbl.Monstop)
    {
        if (key != '3')
            key = 0;
    }
}
else
{
    key=0;
}

if (NewStatTbl.SCSI_Overflow && !NewStatTbl.Monstop) /* check for scsi of */
{
    NewStatTbl.SCSI_Overflow = 0;
    if(StopMon())
    {
        NewStatTbl.Monstop=1;
        screenmsg("SCSI Buffer Overflow, Monitor Stopped!", "\a", 0);
    }
    else
    {
        screenmsg("SCSI Buffer Overflow, but error Stopping Monitor", "\a", 0);
    }
}

if (NewStatTbl.SCSI1_Full || NewStatTbl.SCSI2_Full) /* SCSI data yet. */
{
    switch(Oldest_SCSI_Buffer)
    {
        case(1): Write_Stat = write(Data_FH, SCSI_Buffer1, DATA_LENGTH);
                 NewStatTbl.SCSI1_Full = 0;

                 NewStatTbl.Reception=!NewStatTbl.Reception;

                 if (Write_Stat != DATA_LENGTH)
                 {
                     StopMon();
                     screenmsg("File Write Error, Monitor Stopped!", "\a", 0);
                     NewStatTbl.Monstop=1;
                     NewStatTbl.File_Error=1;
                     refreshstatus();
                 }
                 else

```

```

        NewStatTbl.bytecount = NewStatTbl.bytecount + DATA_LENGTH;

if(NewStatTbl.SCSI2_Full)
{
    Write_Stat = write(Data_FH, SCSI_Buffer2, DATA_LENGTH);
    NewStatTbl.SCSI2_Full = 0;

    NewStatTbl.Reception=!NewStatTbl.Reception;

    if (Write_Stat != DATA_LENGTH)
    {
        StopMon();
        screenmsg("File Write Error, Monitor Stopped!", "\a",
            0);
        NewStatTbl.Monstop=1;
        NewStatTbl.File_Error=1;
        refreshstatus();
    }
    else
        NewStatTbl.bytecount = NewStatTbl.bytecount +
            DATA_LENGTH;
}
break;

case(2): Write_Stat = write(Data_FH, SCSI_Buffer2, DATA_LENGTH);
NewStatTbl.SCSI2_Full = 0;

NewStatTbl.Reception=!NewStatTbl.Reception;

if (Write_Stat != DATA_LENGTH)
{
    StopMon();
    screenmsg("File Write Error, Monitor Stopped!", "\a", 0);
    NewStatTbl.Monstop=1;
    NewStatTbl.File_Error=1;
    refreshstatus();
}
else
    NewStatTbl.bytecount = NewStatTbl.bytecount + DATA_LENGTH;

if(NewStatTbl.SCSI1_Full)
{
    Write_Stat = write(Data_FH, SCSI_Buffer1, DATA_LENGTH);
    NewStatTbl.SCSI1_Full = 0;

    NewStatTbl.Reception=!NewStatTbl.Reception;

    if (Write_Stat != DATA_LENGTH)
    {
        StopMon();
        screenmsg("File Write Error, Monitor Stopped!", "\a",
            0);
        NewStatTbl.Monstop=1;
        NewStatTbl.File_Error=1;
        refreshstatus();
    }
}

```

```

        else
            NewStatTbl.bytecount = NewStatTbl.bytecount +
                DATA_LENGTH;
    }
    break;

    }
}

/* if (crashtesten)
{
    if (crashtest)          /* This test code is to help identify when */
    {                      /* a crash of the hardware occurs. */
        crashtest = 0;
        crashcount = 0;    /* if int routine set test, then reset test */
    }
    else
    {
        crashcount++;      /* if int has not reacted then inc. count */
    }

    if (crashcount == CRASHTHRESHOLD)
    {
        crashtesten = 0;
        crashcount = 0;
        screenmsg("Possible Hardware Crash.", "\a\a\a\a", 0);
    }
} */

}while(1); /* end of event loop */
}          /* ending for main */

/*****
getyesno - routine to get either yes or no from the user
          returned values: 1=yes and 0=no
*****/

int getyesno(void)
{
    int key;

    do
    {
        screenmsg("1 for Yes, 0 for No", "", 0);
        key = getch();
    }
    while ((key != '0') && (key != '1'));
    printf(" %c", (char)key);

    return(key-0x30); /* returns ASCII value -30h
}
*/

```

```

/*****
    printhead - prints the header to the screen. The header is choice of
                commands for the lbi program. Uses ANSI escape sequences
                for cursor control. ANSI.SYS driver must be enabled
                in the config.sys file with the statement:

                        DEVICE=path\ANSI.SYS

*****/

void printhead(void)
{
    printf("\033[1;1H ----Monitor Interface Commands-----");
    printf("\033[3;1H 1  WRITE Level/Wait Information");
    printf("\033[4;1H 2  START_Monitor");
    printf("\033[5;1H 3  STOP_Monitor");
    printf("\033[6;1H 4  RESET_Monitor");
    printf("\033[7;1H 5  GET_Monitor Status");
    printf("\033[8;1H 6  REDRAW Screen");
    printf("\033[9;1H 7  Exit to DOS");
    printf("\033[11;1H -----");
    printf("\033[12;1H Enter command number: ");

    printf("\033[15;1H -----Messages-----");

    printf("\033[1;40H -----Status-----");
    printf("\033[15;40H -----Header Information-----");

    printf("\033[10;40H Bytes Received: ");

    printf("\033[12;23H");          /* sets cursor for command */
}

/*****
    screenmsg - prints the message string passed to the routine to the screen
                in the message location.
*****/

void screenmsg(char * string, char * cntstring, int param_in)
{
    static char *string1, *string2, *string3, *string4;
    static param[4];

    string1=string2;          /* string rotation section */
    string2=string3;
    string3=string4;
    string4=string;          /* set new string */

    param[1]=param[2];        /* parameter rotation section */
    param[2]=param[3];        /* parameter rotation section */
    param[3]=param[4];        /* parameter rotation section */
    param[4]=param_in;        /* parameter rotation section */

    printf("\033[17;0H\033[K"); /* clear to end of message line 1 */
    printf("\033[19;0H\033[K"); /* clear to end of message line 2 */
    printf("\033[21;0H\033[K"); /* clear to end of message line 3 */
}

```



```

printf("\033[23;0H\033[K");          /* clear to end of message line 4      */

printf("\033[17;2H");
printf(string1, param[1]);            /* print message                      */

printf("\033[19;2H");
printf(string2, param[2]);            /* print message                      */

printf("\033[21;2H");
printf(string3, param[3]);            /* print message                      */

printf("\033[23;2H");
printf(string4, param[4]);

printf("%s", cntstring);              /* prints controls for latest msg     */

printf("\033[24;2H\033[K");          /* sets cursor for clearing           */
}

/*****
    refreshstatus - reads various status flags and will print them to
                    the screen in various places in the status area.
*****/

void refreshstatus(void)
{
    int i;

    if(OldStatTbl.Mongo && !OldStatTbl.Monstop)    /* if monitor is running then do */
    {                                                /* commands below */
        if(OldStatTbl.Reception != NewStatTbl.Reception)
        {
            OldStatTbl.Reception      = NewStatTbl.Reception;
        }

        if(OldStatTbl.bytecount != NewStatTbl.bytecount)
        {
            printf("\033[10;56H %lu", NewStatTbl.bytecount);
            OldStatTbl.bytecount      = NewStatTbl.bytecount;
        }

        if(OldStatTbl.Monstop != NewStatTbl.Monstop)
        {
            OldStatTbl.Mongo = 0;    /* by changing Mongo to 0, I am effectively */
            /* exiting the if then statement when refresh */
            /* is called again, and allowing the values of */
            /* Mongo and Monstop to be displayed. */
        }
    }
    else    /* if monitor not running then perform these */
    {
        if(OldStatTbl.Acked != NewStatTbl.Acked)
        {
            OldStatTbl.Acked      = NewStatTbl.Acked;

            if(NewStatTbl.Acked)
                printf("\033[3;40H Monitor Online!      ");
        }
    }
}

```

```

else
    printf("\033[3;40H Monitor not manually acknowledged!");
}

if(OldStatTbl.Cntrl_Val != NewStatTbl.Cntrl_Val)
{
    printf("\033[4;40H Control Value is %X", NewStatTbl.Cntrl_Val);
    OldStatTbl.Cntrl_Val = NewStatTbl.Cntrl_Val;
}

if(OldStatTbl.Monstop != NewStatTbl.Monstop
    || OldStatTbl.Mongo != NewStatTbl.Mongo)
{
    OldStatTbl.Mongo = NewStatTbl.Mongo;
    OldStatTbl.Monstop = NewStatTbl.Monstop;

    if(NewStatTbl.Mongo && !NewStatTbl.Monstop)
        printf("\033[5;40H Monitor Started!  ");
    else
    {
        if(NewStatTbl.Mongo && NewStatTbl.Monstop)
            printf("\033[5;40H Monitor Stopped!");
        else
            printf("\033[5;40H Monitor Reset!  ");
    }
}

if(OldStatTbl.SCSI_Error != NewStatTbl.SCSI_Error)
{
    printf("\033[7;40H The Last SCSI error was %i", NewStatTbl.SCSI_Error);
    OldStatTbl.SCSI_Error = NewStatTbl.SCSI_Error;
}

if(OldStatTbl.SCSI_Overflow != NewStatTbl.SCSI_Overflow)
{
    printf("\033[8;40H SCSI_Buffer has Overflowed!!!");
    OldStatTbl.SCSI_Overflow = NewStatTbl.SCSI_Overflow;
}

if(OldStatTbl.File_Error != NewStatTbl.File_Error)
{
    printf("\033[9;40H A file error has occurred!!!");
    OldStatTbl.File_Error = NewStatTbl.File_Error;
}

if(updatefname)
{
    printf("\033[12;40H Current file is %s          ", fname);
    updatefname = 0;
}

}

/* Update the old status table with new status values */

OldStatTbl.MonStat = NewStatTbl.MonStat;
OldStatTbl.SCSI1_Full = NewStatTbl.SCSI1_Full;

```

```

OldStatTbl.SCSI2_Full    = NewStatTbl.SCSI2_Full;
OldStatTbl.Last_Usr_CMD  = NewStatTbl.Last_Usr_CMD;

printf("\033[12;23H");          /* sets cursor for command */
}

/*****

getheader - subroutine to get and display the file header for the
currently active file.  This data will be displayed
in the lower right part of the screen.

This routine returns the array with the header string
in place.  getheader also initializes the first 256
bytes for the file.  The main loop simply has to
write the first 256 bytes out to the file, without
using pointer arithmetic.

*****/

char * getheader(void)
{
    char charheader[72]; /* initialize the return array */
    char tempstring[37];  /* temporary string for writing */
    int i;                /* i is loop var. for string op. */

    for(i = 0; i <= 71; i++)
        charheader[i] = 0x20;          /* init charheader to spaces */

    tempstring[36] = 0x00;              /* set end of string marker on tempstring */

    screenmsg("Add what file information? ", "", 0);
    screenmsg("Max. of 72 characters. ", "", 0);

    gets(charheader);                  /* retrieve the header information */

    for (i = 0; i <= 71; i++)          /* deletes the null from the array */
    {
        if(charheader[i] == 0)
            charheader[i] = 0x20;
    }

    for (i = 0; i <= 35; i++)          /* loop for xfering first 34 characters */
    {
        tempstring[i] = charheader[i]; /* xfer first 34 of charheader to tempstring */
        fullheader[i] = charheader[i]; /* xfer first 34 of charheader to fullheader */
    }

    printf("\033[18;41H\033[K");      /* set cursor and clear print region */
    printf("%s", tempstring);          /* print first part of header information */

    for (i = 36; i <= 71; i++)
    {
        tempstring[i-36] = charheader[i]; /* xfer 35-68 of charheader to tempstring */
    }
}

```

```

    fullheader[i] = charheader[i];    /* xfer 35-68 of charheader to fullheader */
}

printf("\033[20;41H\033[K");          /* sets cursor and clear to end of line 20 */
printf("%s", tempstring);             /* prints second half of charheader */

for(i = 72; i <= 104; i++)            /* this section places the FFFE values into
fullheader */
{
    if (i%2 == 1)
        fullheader[i] = 0xFE;        /* if i odd then poke FE */
    else
        fullheader[i] = 0xFF;        /* if i even then poke FF */
}

for(i = 104; i <= 255; i++)          /* test code - remove in final */
{
    fullheader[i] = 0xAA;            /* test code - remove in final */
}
} /* close for getheader */

```

```

/*****

```

```

    SetCntrMon - Sets the control values on the Monitor. Bit 0 of
                  Cntrl_Val determines Level Capture, 1=capture levels
                  0=no levels captured. Bit 1 of Cntrl_Val determines
                  Wait Captures, with 1=capture wait events and
                  0=no wait events captured. This program returns a
                  0 if no error occurred, and if an error occurred, the
                  program will return the SCSI bus error that occurred.

```

```

*****/

```

```

int SetCntrMon(int CV)
{
    Command.OP_Code = SET_CONTROL;      /* Sets SCSI CMD */
    Out_Buff[0] = (char)CV;            /* Sets SCSI Data, casting int */
    OEM_Tbl.Expect_Length = 1;

    CSMSCSI(&OEM_Tbl);                /* Calls SCSI Initiator Routine */
    Out_Buff[0] = 0xFF;                /* resets the Out_Buff */

    return(OEM_Tbl.Error_Code);
}

```

```

/*****

```

```

    StartMon - Tells the Monitor to start recording information
               according to the control pattern that should have
               already been written by the SetCntrMon routine. This
               program returns a 0 if no error occurred, and if
               an error occurred, the program will return the SCSI
               bus error.

```

```

*****/

```

```

int StartMon(void)
{
    Command.OP_Code = START_MONITOR;          /* Set Op-Code for SCSI Command */
    OEM_Tbl.Expect_Length = 1;                /* Set Data Length */

    CSMSCSI(&OEM_Tbl);                        /* Attempt a SCSI transfer */

    return(OEM_Tbl.Error_Code);               /* Return SCSI error if any */
}

```

```

/*****

```

```

    StopMon - Tells the Monitor to stop recording information
              This program returns a 0 if no error occurred, and if
              an error occurred, the program will return the SCSI
              bus error.

```

```

*****/

```

```

int StopMon(void)
{
    Command.OP_Code = STOP_MONITOR;          /* Set Op-Code for SCSI Command */
    OEM_Tbl.Expect_Length = 1;                /* Set SCSI Data Length */

    CSMSCSI(&OEM_Tbl);                        /* Attempt a SCSI Transfer */

    return(OEM_Tbl.Error_Code);              /* return SCSI error if any */
}

```

```

/*****

```

```

    ResetMon - Tells the Monitor to reset to default values, and
              stop recording information, if monitoring was occurring
              This program returns a 0 if no error occurred, and if
              an error occurred, the program will return the SCSI
              bus error.

```

```

*****/

```

```

int ResetMon(void)
{
    Command.OP_Code = RESET_MONITOR;         /* Set SCSI Command Op-Code */
    OEM_Tbl.Expect_Length = 1;                /* Set Data Length for transfer */
    CSMSCSI(&OEM_Tbl);

    return(OEM_Tbl.Error_Code);
}

```

```

/*****

```

```

    AckMon - This routine will simply confirm the connection
             between the PS/2 and the Monitor. This
             program returns a 0 if no error occurred, and if
             an error occurred, the program will return the SCSI
             bus error.

```

```

*****/

int AckMon(void)
{
    Command.OP_Code = ACK_MONITOR;
    OEM_Tbl.Expect_Length = 1;
    CSMSCSI(&OEM_Tbl);

    return(OEM_Tbl.Error_Code);
}

/*****

RetrieveStatus - This program will return the status of the monitor
                at the time of the command. The current status
                indicators are
                    00    No error on Monitor
                    FF    FIFO Overflow has occurred.
                This program returns a an unsigned integer value that
                has the lower byte set to Status returned by the
                SCSI program and the upper byte set to the Error Code
                returned by the SCSI program. If no error occurred
                then the upper byte will be set to 00h and the lower
                byte will be set to the status returned by the
                Monitor. If an error occurred then the upper byte
                will have the SCSI Error Code and the lower byte
                will be indeterminate. In other words check the
                high byte to see if the low byte is acceptable data.

*****/

long RetrieveStatus(void)
{
    Command.OP_Code = SEND_STATUS;
    OEM_Tbl.Expect_Length = 1;
    CSMSCSI(&OEM_Tbl);

    return((long)OEM_Tbl.Status | ((long)OEM_Tbl.Error_Code << 16));
}

/*****

InitOEMTable - Initializes the OEM Table, the Command buffer, and
              the data buffer for use in the SCSI Initiator
              routine.

*****/

void InitOEMTable(void)
{
    OEM_Tbl.CMDBlk_addr = (CMD_Block far *) &Command;
    OEM_Tbl.DATABlk_addr = (uchar far *) Out_Buff;
    OEM_Tbl.Expect_Length = 10;
    OEM_Tbl.Actual_Length = 0;
    OEM_Tbl.Timeout = 1;
    OEM_Tbl.Blk_A = 0x8001; /* Fix 1 on CSM SCSI code */
}

```

```

OEM_Tbl.Blk_B      = 0;
OEM_Tbl.Status     = 0;
OEM_Tbl.MSG        = 0;
OEM_Tbl.Control_Lines = 0;
OEM_Tbl.SCSI_ID    = 7;
OEM_Tbl.Parity     = 0;
OEM_Tbl.Error_Code = 0;

Command.OP_Code = 0xFF;
Command.LUN     = 0x00;
Command.MSB     = 0xAA;
Command.LBA     = 0x55;
Command.LSB     = 0x37;
Command.Control = 0xB6;

Out_Buff[0] = 0xFF;
Out_Buff[1] = 0x00;
}

/* ----- Monitor Reaction Functions ----- */

/*****

Target01_Routine - Part of the CSM Target Table for SCSI
                  reaction.

This Routine responds to hex 01 from the monitor, or the
Receive_Data command. This routine will check to see
if the SCSI_Data_Buffer is full, and if so it will
then set the SCSI_Buffer_Overflow flag. If the buffer
is empty then the routine will receive DATA_LENGTH of bytes
from the monitor and put them into the SCSI_Buffer1.

Routines written by Ken Baker. Adapted and expanded from
code written by Robert Thompson. Nov. 5, 1990!

*****/

void far Target01_Routine(char *cmdbytes, char *msgbyte, int Init_ID)
{
    switch(Oldest_SCSI_Buffer)
    {
        case(1): RECV_DATA(SCSI_Buffer1, (long)DATA_LENGTH, DMA_FLAG);
                 BUS_FREE();

                 if(NewStatTbl.SCSI1_Full)
                     NewStatTbl.SCSI_Overflow = 1;
                 else
                 {
                     NewStatTbl.SCSI1_Full = 1;

                     if(NewStatTbl.SCSI2_Full)
                         Oldest_SCSI_Buffer = 2;
                     else
                         Oldest_SCSI_Buffer = 1;
                 }
    }
}

```

```

        break;

    case(2): RECV_DATA(SCSI_Buffer2, (long)DATA_LENGTH, DMA_FLAG);
             BUS_FREE();

             if(NewStatTbl.SCSI2_Full)
                 NewStatTbl.SCSI_Overflow = 1;
             else
             {
                 NewStatTbl.SCSI2_Full = 1;

                 if(NewStatTbl.SCSI1_Full)
                     Oldest_SCSI_Buffer = 1;
                 else
                     Oldest_SCSI_Buffer = 2;
             }
             break;
    }
}

/*****

Target00_Routine - Part of the CSM Target Table for SCSI
                  reaction.

This Routine responds to hex 00 from the monitor, or the
Ack_Connection command.

Operation: This command simply accepts a command byte
and then will send the numbers 00 55 AA and FF to the
monitor. The monitor will then know the connection
with the host system has been established.

Routine written by Ken Baker. Nov. 6, 1990!

*****/

void far Target00_Routine(char *cmdbytes, char *msgbyte, int Init_ID)
{
    #define SEND_BUFF_SIZE 4

    char Send_Buffer[SEND_BUFF_SIZE];          /* send buffer array defn. */

    Send_Buffer[0]=0x00;                        /* loading send buffer */
    Send_Buffer[1]=0x55;
    Send_Buffer[2]=0xAA;
    Send_Buffer[3]=0xFF;

    SEND_DATA(Send_Buffer, (long)SEND_BUFF_SIZE, 0); /* prog. IO SCSI send */
    BUS_FREE();
}

```


LBFV107

```

( Start )

    HEX
    2000 TIB !
    50 TIB 2 + !

( LBF - LOGIC BOX FIRMWARE v1.06 )
( DIFFERENT FROM V1.00 AS I HAVE ADDED CODE FOR PRINTING IN
( MAIN, WHICH ADDS A NICE HEADER.

( DIFFERENT FORM V1.01 AS I HAVE ADDED CODE FOR
( THE FINAL SYSTEM
( WHICH WILL INCORPORATE THE DMA CONTROLLER AND
( COMMUNICATION
( WITH THE HOST SYSTEM

( VERSION 1.3 HAS DIFFERENCES IN THE INTERRUPT STRUCTURE
( FOR THE SYSTEM. THE SCSI BUS HAS BEEN ASSIGNED TO THE
( XIRQ ROUTINE AND CODE HAS BEEN ADDED TO THE DMA SERVICE
( ROUTINE THAT WILL BLOCK OTHER INTERRUPT SOURCES.

( in v1.4 I have added code supporting status checking when
( the host requests it

( by Ken Baker )
( MAR. 12, 1991 )

( LOGIC BOX FIRMWARE FOR THE UTUCAM SYSTEM.
( THIS FIRMWARE CONTROLS ALL ACTIVITY ON THE MONITOR
( WHICH IS THE SEPARATE HARDWARE BOX FOR MONITORING THE
( 4700 SIGNALS.

( MEMORY MAP - )
( SCSI REGISTERS 5000h to 5399h )
( SYSTEM RAM 2000h TO 3FFFh )
( EPROM C000h to DFFFh )

( Assembler Download - Will be discarded after PROMing code )

    2050 DP !

( ASM6811 )

( VERSION 1.9 06/01/89 )
( : CODE-SUB [COMPILE] CODE-SUB [COMPILE] ASSEMBLER ; IMMEDIATE )

( NOTE: VARIABLE MODE NEEDS TO BE ASSIGNED TO RAM IF THIS TO BE EPROM/ROM'ed )
( IE: 8 CONSTANT MODE )
VARIABLE MODE
: # 00 MODE ! ;
: DIR 10 MODE ! ;
: ,X 20 MODE ! ;
: ,Y 120 MODE ! ;
: EXT 30 MODE ! ;
EXT ( INITIALIZE MODE VARIABLE )

```

```

: ?# MODE @ 0 = ;
: ?DIR MODE @ 10 = ;
: ?,X MODE @ 20 = ;
: ?,Y MODE @ 120 = ;
: ?EXT MODE @ 30 = ;
: MODE-LSB MODE @ FF AND ;
: ERROR EXT ' ID. CFA 4A + EXECUTE ;
: RANGE-C, DUP FF00 AND IF HERE 1+ - DUP ABS FF00 AND
                                IF 3 ERROR THEN THEN C, ;

: CPU <BUILDS C, DOES> C@ C, EXT ; ( SINGLE BYTE OP-CODE )
: PG-2-CPU <BUILDS C, DOES> 18 C, C@ C, EXT ; ( 18 AND SINGLE BYTE OP-CODE )

: HHLL/LL, ?EXT IF , ELSE C, THEN EXT ;
: 2HHLL/LL, ?EXT ?# OR IF , ELSE C, THEN EXT ;
: MODE-ADJ, ( a n --- a )
  ?EXT IF OVER FF00 AND 0= IF DIR THEN THEN MODE-LSB OR C, ;
: 18,Y ?,Y IF 18 C, THEN ;
: SOK? ?# IF 3 ERROR THEN ;

: xx12-CPU <BUILDS C, DOES> 18,Y ?# ?DIR OR IF 3 ERROR THEN
                                C@ MODE-LSB OR C, HHLL/LL, ;
: 1112-CPU <BUILDS C, DOES> 18,Y C@ MODE-ADJ, HHLL/LL, ;
: x112-CPU <BUILDS C, DOES> 18,Y SOK? C@ MODE-ADJ, HHLL/LL, ;
: 2112-CPU <BUILDS C, DOES> 18,Y C@ MODE-ADJ, 2HHLL/LL, ;

: (OP-DD-MM) 18,Y MODE-LSB IF C OR THEN C, C, C, ;
: OP-RR <BUILDS C, DOES> C@ C, RANGE-C, EXT ;
: OP-DD-MM <BUILDS C, DOES> C@ (OP-DD-MM) EXT ;
: OP-DD-MM-RR <BUILDS C, DOES> C@ (OP-DD-MM) RANGE-C, EXT ;

: CPX, ?,Y IF CD C, THEN 8C MODE-ADJ, 2HHLL/LL, ;
: LDX, ?,Y IF CD C, THEN CE MODE-ADJ, 2HHLL/LL, ;
: STX, ?,Y IF CD C, THEN SOK? CF MODE-ADJ, HHLL/LL, ;
: CPY, ?,X IF 1A ELSE 18 THEN C, 8C MODE-ADJ, 2HHLL/LL, ;
: LDY, ?,X IF 1A ELSE 18 THEN C, CE MODE-ADJ, 2HHLL/LL, ;
: STY, ?,X IF 1A ELSE 18 THEN C, SOK? CF MODE-ADJ, HHLL/LL, ;
: CPD, ?,Y IF CD C, ELSE 1A C, THEN 83 MODE-ADJ, 2HHLL/LL, ;

00 CPU TEST,
01 CPU NOP,
02 CPU IDIV,
03 CPU FDIV,
04 CPU LSRD,
05 CPU ASLD, 05 CPU LSLD,
06 CPU TAP,
07 CPU TPA,
08 CPU INX,
09 CPU DEX,
0A CPU CLV,
0B CPU SEV,
0C CPU CLC,
0D CPU SEC,
0E CPU CLI,
0F CPU SEI,

```

10 CPU SBA,
11 CPU CBA,
12 OP-DD-MM-RR BRSET,
13 OP-DD-MM-RR BRCLR,
14 OP-DD-MM BSET,
15 OP-DD-MM BCLR,
16 CPU TAB,
17 CPU TBA,
(18 PAGE 2)
19 CPU DAA,
(1A PAGE 3)
1B CPU ABA,
(1C BSET,
(1D BCLR,
(1E BRSET,
(1F BRCLR,
20 OP-RR BRA,
21 OP-RR BRN,
22 OP-RR BHI,
23 OP-RR BLS,
24 OP-RR BCC,
24 OP-RR BHS,
25 OP-RR BCS,
25 OP-RR BLO,
26 OP-RR BNE,
27 OP-RR BEQ,
28 OP-RR BVC,
29 OP-RR BVS,
2A OP-RR BPL,
2B OP-RR BMI,
2C OP-RR BGE,
2D OP-RR BLT,
2E OP-RR BGT,
2F OP-RR BLE,
30 CPU TSX,
31 CPU INS,
32 CPU PULA,
33 CPU PULB,
34 CPU DES,
35 CPU TXS,
36 CPU PSHA,
37 CPU PSHB,
38 CPU PULX,
39 CPU RTS,
3A CPU ABX,
3B CPU RTI,
3C CPU PSHX,
3D CPU MUL,
3E CPU WAI,
3F CPU SWI,
40 CPU NEGA,
(41 NC)
(42 NC)
43 CPU COMA,
44 CPU LSRA,
(45 NC)

```

46 CPU RORA,
47 CPU ASRA,
48 CPU ASLA,
49 CPU ROLA,
4A CPU DECA,
( 4B NC )
4C CPU INCA,
4D CPU TSTA,
( 4E NC )
4F CPU CLRA,
50 CPU NEGB,
( 51 NC )
( 52 NC )
53 CPU COMB,
54 CPU LSRB,
( 55 NC )
56 CPU RORB,
57 CPU ASRB,
58 CPU ASLB,
59 CPU ROLB,
5A CPU DECB,
( 5B NC )
5C CPU INCB,
5D CPU TSTB,
( 5E NC )
5F CPU CLRB,
( 60-7F )
40 xx12-CPU NEG,
43 xx12-CPU COM,
44 xx12-CPU LSR,
46 xx12-CPU ROR,
47 xx12-CPU ASR,
48 xx12-CPU ASL,
49 xx12-CPU ROL,
4A xx12-CPU DEC,
4C xx12-CPU INC,
4D xx12-CPU TST,
4E xx12-CPU JMP,
4F xx12-CPU CLR,
( 80-BF )
80 1112-CPU SUBA,
81 1112-CPU CMPA,
82 1112-CPU SBCA,
83 2112-CPU SUBD,
84 1112-CPU ANDA,
85 1112-CPU BITA,
86 1112-CPU LDAA,
87 x112-CPU STAA,
88 1112-CPU EORA,
89 1112-CPU ADCA,
8A 1112-CPU ORAA,
8B 1112-CPU ADDA,
( 8C CPX,
8D x112-CPU JSR,
8E 2112-CPU LDS,
8F x112-CPU STS,

```

```

8F CPU XGDY,
( C0-FF )
C0 1112-CPU SUBB,
C1 1112-CPU CMPB,
C2 1112-CPU SBCB,
C3 2112-CPU ADDD,
C4 1112-CPU ANDB,
C5 1112-CPU BITB,
C6 1112-CPU LDAB,
C7 x112-CPU STAB,
C8 1112-CPU EORB,
C9 1112-CPU ADCB,
CA 1112-CPU ORAB,
CB 1112-CPU ADDB,
CC 2112-CPU LDD,
CD x112-CPU STD,
( CE LDX, )
( CF STX, )
CF CPU STOP,

```

```

08 PG-2-CPU INY,
09 PG-2-CPU DEY,
30 PG-2-CPU TSY,
35 PG-2-CPU TYS,
38 PG-2-CPU PULY,
3A PG-2-CPU ABY,
3C PG-2-CPU PSHY,

```

```

8F PG-2-CPU XGDY,

```

```

8D OP-RR BSR,

```

```

: TOP ,Y    0 ;      ( ADDRESS THE BOTTOM OF THE STACK *)
: SEC ,Y    2 ;      ( ADDRESS SECOND ITEM ON STACK *)

```

```

: ?EXEC STATE @ IF 12 ERROR THEN ;
: ?PAIRS - IF 13 ERROR THEN ;

```

```

: BEGIN,    HERE 1 ;
: UNTIL,    ?EXEC >R 1 ?PAIRS R> C, HERE 1+ - C, ;
: AGAIN,    20 UNTIL, ;
: IF,       C, HERE 0 C, 2 ;
: THEN,     ?EXEC 2 ?PAIRS HERE OVER 1+ - SWAP C! ;
: ELSE,     2 ?PAIRS HERE 1+ 0 BRA,
            SWAP HERE OVER 1+ - SWAP C! 2 ;
: .NOT. 1 XOR ;      ( REVERSE ASSEMBLY TEST )

```

```

20 CONSTANT .FL.
21 CONSTANT .TR.
22 CONSTANT .LS.
23 CONSTANT .HI.
24 CONSTANT .CS.
24 CONSTANT .LO.
25 CONSTANT .CC.
25 CONSTANT .HS.

```

```

26 CONSTANT .EQ.
27 CONSTANT .NE.
28 CONSTANT .VS.
29 CONSTANT .VC.
2A CONSTANT .--.
2B CONSTANT .++.
2C CONSTANT .LT.
2D CONSTANT .GE.
2E CONSTANT .LE.
2F CONSTANT .GT.

```

```

: BIT-BR <BUILDS C, DOES> C@ 18,Y MODE-LSB IF C OR THEN C, C, ; IMMEDIATE
12 BIT-BR .CLR.
13 BIT-BR .SET.

```

```

' @ CFA FE43 FE22 - + CONSTANT PUSH
' @ CFA FE47 FE22 - + CONSTANT NEXTSD
' @ CFA FE47 FE22 - + CONSTANT PUT
' @ CFA FE4A FE22 - + CONSTANT NEXT
' @ CFA FE4C FE22 - + CONSTANT NEXT3
' @ CFA FE50 FE22 - + CONSTANT NEXT1
' @ CFA FE52 FE22 - + CONSTANT NEXT2
' 1+ CFA FC97 FC7C - + CONSTANT POP
' 1+ CFA FC93 FC7C - + CONSTANT POPTWO

```

```

0 CONSTANT W
2 CONSTANT IP
4 CONSTANT UP

```

```

( MARK AND RESOLVE ROUTINES FOR USE OF POINTERS
( WRITTEN AND ADDED TO ASM6811 BY KEN BAKER

```

```

: >MARK-BYTE DP @ ;
: >RESOLVE-BYTE DUP DP @ SWAP - DUP FF > IF ERROR THEN SWAP 1 - C! ;
: <MARK-BYTE DP @ ;
: <RESOLVE-BYTE DP @ - DUP FF80 < IF ERROR THEN CR DP @ 1 - C! ;

```

```

( MACRO DEFINITIONS )

```

```

: PUTD, DEY, DEY, TOP STD, ; ( MACROS, FORTH DATA STACK OPS )
: POPD, TOP LDD, INY, INY, ;
: PUTX, DEY, DEY, TOP STX, ;
: POPX, TOP LDX, INY, INY, ;

```

```

: SAVE-REGS, PSHB, PSHA, PSXH, ; ( MACROS, SAVES AND RESTORS REG. )
: RESTORE-REGS, PULX, PULA, PULB, ;

```

```

( ***** )
( BEGINNING OF UTUCAM MONITOR CODE )
( ***** )

```

```

( BEGINNING OF RAM )

```

```

( RAM VARIABLE Formation )

```

```

VARIABLE RAM_POINTER          ( POINTER FOR CREATING VAR.

HWORD
: IS CONSTANT ;
HWORD

3800 RAM_POINTER !           ( SET VARIABLES AT 3800H

: RAM                          ( WORDS FOR FORMING VAR.
  RAM_POINTER @
  SWAP
  RAM_POINTER +! ;

( BEGINNING OF PROM -- VARIABLES AND CONSTANT FORMATION )

C000 DP !

( VARIABLE FORMATION )

  1 RAM IS SCSI_BUFF_FULL?    ( 1=SCSI Data Received )
  8 RAM IS SCSI_BUFFER        ( INIT ID, 6 CMD bytes and 1 DATA bytes )
  1 RAM IS SCSI_BUFF_OVFLW    ( 1=SCSI BUFFER OVERFLOWED, BAD )
  2 RAM IS 4.1COUNT          ( INTERVAL TIMER VARIABLE )
  1 RAM IS STATUS             ( STATUS BYTE
  2 RAM IS SCRATCH            ( SCRATCH PAD AREA )

( CONSTANT FORMATION )

B000 CONSTANT PORTA           ( DEF'N OF PORT A )
B008 CONSTANT PORTD           ( DEF'N OF PORT D ADDRESS )
B009 CONSTANT DDRD            ( DEF'N OF DATA DIR. REG FOR D)
B024 CONSTANT TMSK2           ( TIMER MASK 2 DEF'N )
B025 CONSTANT TFLG2           ( TIMER FLAG 2 DEF'N )
B026 CONSTANT PACTL           ( DEF'N OF PULSE ACC. CNTRL REG. )
5000 CONSTANT SCSI_ADO        ( ADDRESS FOR SCSI REG. )

0   CONSTANT DATA-PHASE      ( DEF. OF DATA PHASE FOR SCSI COMMANDS )
2   CONSTANT CMD/STAT-PHASE    ( DEF. OF CMD PHASE )
6   CONSTANT MSG-PHASE        ( DEF. OF MESSAGE PHASE FOR SCSI )
80  CONSTANT MON-SCSI-ID      ( SCSI ID FOR MONITOR )
1   CONSTANT PS/2-ID          ( ID FOR CONTROLLING PS/2 )
1   CONSTANT DUMP-DATA        ( DUMP FIFO DATA COMMAND TO PS/2
0   CONSTANT ACK-CONNECT      ( ACK CONNECTION COMMAND )

( *****
(
( SET-TARGET-MODE - SETS THE TARGET MODE BIT, AND SETS THE ID
( ON THE 5380 SCSI INTERFACE CHIP.
(
( *****

CODE-SUB SET_TARGET_MODE
      SAVE-REGS,      ( SAVE REGISTERS TO STACK )
SCSI_ADO # LDY,      ( SET POINTER TO SCSI REGS. )
,X 2 LDAA,      ( LOAD ACC. A WITH INITIATOR CMD REG. )
40 # ORAA,      ( GET VALUE FOR TARGET MODE )

```



```

,X 2      STAA,      ( SET TARGET MODE )
MON-SCSI-ID # LDAA,      ( GET OUR ID FOR SELECTION ID REG. )
,X 4      STAA,      ( SET ID SELECT REG. )
          RESTORE-REGS, ( RESTORE REGS. FROM STACK )
          RTS,

END-CODE

( *****
(
( INTON - ROUTINE TO CLEAR THE I-BIT AND THE X BIT
(      IN THE CONDITION CODE REG.
(
( THIS WILL ALLOW ANY UNMASKED INTERRUPTS TO OCCUR
(
( *****

CODE-SUB INTON
      0 #      LDAA,
              TAP,      ( ENABLES IRQ AND XIRQ )
              RTS,

END-CODE

( *****
( INTOFF - ROUTINE TO SET THE I-BIT IN THE CONDITION CODE REG.
(
( THIS WILL DIS-ALLOW ANY NONMASKABLE INTERRUPTS TO OCCUR
(
( *****

CODE-SUB INTOFF
          SEI,      ( SETS INT BIT, INHIBITING INT. )
          RTS,

END-CODE

( *****
( ARB/SELECT - GOES THROUGH THE ARBITRATION AND SELECTION
( SCSI PHASES.
(
( ASSUMES THAT THE MONITOR ID IS ON TOP OF STACK, WITH TARGET
( ID NEXT ON STACK.
(
( TOS - TARGET ID
(
( THIS ROUTINE MUST HAVE INITIATOR MODE SET BY THE CALLING
( ROUTINE. IF NOT, THEN THE SYSTEM WILL RESPOND TO ITS
( OWN CALL FOR SELECTION.
(
( THIS ROUTINE WILL RETURN AN ERROR CODE.
(      00 = NO ERROR
(      01 = SELECTION TIMEOUT ERROR.
(
( *****

CODE-SUB ARB/SELECT
          SAVE-REGS, ( MACRO TO SAVE A,B,X,Y TO STACK )
SCSI_ADO #  LDX,      ( LDX WITH FIRST SCSI REGISTER

```

```

        CLRA,      ( CLEARS A FOR INIT TARGET CMD REG
,X 3          STAA,      ( SETS TARGET CMD REG FOR BUS CONTROL
MON-SCSI-ID # LDAB,      ( GET OUR ID
,X 0          STAB,      ( SET ID TO SCSI DATA BUS
,X 1 2        BSET,      ( SET ARBITRATE BIT
,X FC 40 1    BRCLR,     ( WAITS FOR ARB IN PROGRESS BIT
                NOP,      ( NOPs FOR DELAY OF 3US
                NOP,
                NOP,
,X F2 20 1    BRSET,     ( BRANCHES IF ARBITRATION LOST
9 #           LDAA,      ( GET READY TO ASSERT BSY
,X 1          STAA,      ( ASSERT BSY
,X 1 2        BCLR,      ( RESET THE ARBITRATE BIT
                POPD,     ( GET TARGET ID FROM FORTH DATA STACK
MON-SCSI-ID # ORAB,      ( OR TARGET ID WITH OUR ID
,X 0          STAB,      ( ASSERTS OUR ID ORed WITH TARGET ID
5 #           LDAA,      ( READY TO DEASSERT BSY, ASSERT SEL
,X 1          STAA,      ( DEASSERT BSY, ASSERT SEL
2 #           LDD,       ( SET UP FOR TIMEOUT CHECK
4.1COUNT    STD,       ( STORE TIMEOUT VALUE TO VAR.
BEGIN,
,X 4          LDAA,      ( CHECK FOR BSY ACTIVE
40 #          ANDA,      ( PARSE OUT BSY SIGNAL ONLY
40 #          CMPA,      ( SEE IF BSY ACTIVE
                .EQ.      ( IF CHECK EQUAL THEN CONTINUE
IF,
                CLRA,      ( SET A,B TO 0
                CLRB,      ( RETURN 00 ERROR CODE = NO ERROR
                PUTD,
1 #           LDAA,      ( SET A TO VALUE FOR INIT CMD REG
,X 1          STAA,      ( ASSERT DATA BUS AND BSY
                RESTORE-REGS,
                RTS,
THEN,
4.1COUNT    LDD,       ( LOAD 4.1COUNT FOR TIMEOUT CHECK
                .EQ.      ( IF TIMEOUT NOT EXPIRED THEN CONTINUE LOOP
UNTIL,
                CLRA,      ( CLEAR A FOR STACK OP.
01 #          LDAB,      ( SET B FOR 01 ERROR CODE=ERROR OCCURRED
                PUTD,
,X 1          STAA,      ( RESET 5380 VALUES
,X 0          STAA,
                RESTORE-REGS, ( MACRO TO GET X,B,A FROM STACK )
                RTS,
END-CODE

( *****
( T_SEND - PERFORMS ONE SCSI SEND OPERATION IN TARGET MODE.
(
( ASSUMES THAT STACK HAS PHASE INFORMATION ON TOP AND DATA
( BYTE TO SEND NEXT TO TOP. PHASE INFORMATION SET UP AS:
( MSG, C/D, I/O AS BITS 2, 1 AND 0.
( *****

CODE-SUB T_SEND      ( ASSUMES PHASE, DATA ARE ON STACK
                SAVE-REGS, ( MACRO TO SAVE A,B,X,Y TO 6811 STACK )

```

```

SCSI_AD0 # LDX,          ( SETS THE POINTER REG. TO SCSI
                POPD,      ( GET PHASE INFORMATION
01 #          ORAB,        ( SET I/O SIGNAL
,X 03         STAB,        ( SET PHASE REG. WITH PHASE INFO.
                POPD,      ( MACRO TO GET DATA FROM DATA STACK
,X 00         STAB,        ( WRITE DATA TO SCSI BUS REGISTER
09 #          LDAA,        ( GET VALUE TO ASSERT DATA BUS
,X 01         STAA,        ( ASSERT DATA BUS
,X 08 03      BSET,        ( SETS THE REQ BIT IN 5380
,X FC 01 05  BRCLR,        ( WAITS FOR THE ACK BIT TO SET
,X 08 03      BCLR,        ( RESET THE REQ BIT
,X FC 01 05  BRSET,        ( WAITS FOR THE ACK BIT TO RESET
                RESTORE-REGS, ( GETS Y,X,B,A FROM 6811 STACK )
                RTS,        ( RETURN TO CALLER

```

END-CODE

```

( *****
( T_RECEIVE - PERFORMS ONE SCSI RECEIVE OPERATION IN TARGET MODE.
(
( ASSUMES THAT STACK HAS PHASE INFORMATION ON TOP.  PHASE
( INFORMATION SET UP AS : MSG, C/D, I/O AS BITS 2, 1 AND 0.
(
( RETURNS DATA AS INFORMATION ON TOP OF STACK.
( *****

```

CODE-SUB T_RECEIVE

```

                SAVE-REGS,
SCSI_AD0 # LDX,          ( SETS THE POINTER REG. TO SCSI
                POPD,      ( RETRIEVES PHASE INFO FROM STACK
,X 03         STAB,        ( WRITES PHASE INFO TO PHASE REG.
,X 08 03      BSET,        ( SETS THE REQ BIT IN 5380
,X FC 01 05  BRCLR,        ( WAITS FOR THE ACK BIT TO SET
,X 00         LDAB,        ( LOAD B FROM SCSI_AD0
,X 08 03      BCLR,        ( RESET THE REQ BIT
,X FC 01 05  BRSET,        ( WAIT FOR THE ACK TO DEACTIVATE
                CLRA,      ( CLEAR A
                PUTD,      ( PUSH D ONTO THE DATA STACK
                RESTORE-REGS,
                RTS,

```

END-CODE

```

( *****
( I_SEND - PERFORMS ONE SCSI SEND OPERATION IN INITIATOR MODE.
(
( ASSUMES THAT STACK HAS PHASE INFORMATION ON TOP AND DATA
( BYTE TO SEND NEXT TO TOP.  PHASE INFORMATION SET UP AS:
( MSG, C/D, I/O AS BITS 2, 1 AND 0 RESPECTIVELY.
( *****

```

CODE-SUB I_SEND

```

                SAVE-REGS,
SCSI_AD0 # LDX,          ( SET UP POINTER REGISTER
                POPD,      ( PULLS PHASE FROM FORTH DATA STACK
,X 03         STAB,        ( SETS TARGET CMD REG.
FE #         ANDB,        ( SETS THE DIRECTION INFORMATION
BEGIN,

```

```

,X 04      LDAA,          ( LOAD A WITH PHASE REGISTER
                RORA,          ( SHIFT TWICE TO ALLIGN PHASE
                RORA,
07 #      ANDA,          ( PARSE OUT THE PHASE LINES
                CBA,          ( COMPARES CURRENT PHASE VS NEEDED PHASE
                .EQ.

UNTIL,
,X FC 20 04 BRCLR,      ( LOOK FOR REQ, WHEN ACTIVE CONTINUE
                POPD,          ( GET DATA TO SEND
,X 00      STAB,          ( WRITE DATA TO OUTPUT
11 #      LDAA,          ( GET VALUE FOR INIT CMD REG
,X 01      STAA,          ( SET INIT CMD REG.
,X FC 20 04 BRSET,      ( LOOK FOR REQ, WHEN INACTIVE CONTINUE
01 #      LDAA,          ( GET VALUE FOR INIT CMD REG
,X 01      STAA,          ( SET INIT CMD REG.
                RESTORE-REGS,
                RTS,

END-CODE

```

```

( *****
( I_RECEIVE - PERFORMS ONE SCSI RECEIVE OPERATION IN INITIATOR MODE.
(
( ASSUMES THAT STACK HAS PHASE INFORMATION ON TOP.  PHASE
( INFORMATION SET UP AS : MSG, C/D, I/O AS BITS 2, 1 AND 0.
(
( RETURNS DATA AS INFORMATION ON TOP OF STACK.
( *****

```

```

CODE-SUB I_RECEIVE
                SAVE-REGS,
SCSI_ADO # LDY,          ( SETS POINTER SCSI REG.
                POPD,          ( GET PHASE INFO FROM STACK
01 #      ORAB,          ( SET DIRECTION INFORMATION

BEGIN,
,X 04      LDAA,          ( LOAD A WITH PHASE REGISTER
                RORA,          ( SHIFT TWICE TO ALIGN PHASE
                RORA,
07 #      ANDA,          ( PARSE OUT THE PHASE LINES
                CBA,          ( COMPARES PHASE VS NEEDED PHASE
                .EQ.

UNTIL,
,X FC 20 04 BRCLR,      ( LOOK FOR REQ, WHEN ACTIVE CONTINUE
,X 0      LDAB,          ( GET VALUE FROM SCSI BUS
01 #      LDAA,          ( GET VALUE TO ASSERT ACK
,X 01      STAA,          ( ASSERT ACK
,X FC 20 04 BRSET,      ( LOOK FOR /REQ.
00 #      LDAA,          ( SET VALUE TO DEASSERT ACK
,X 01      STAA,          ( DEASSERT ACK
                CLRA,          ( CLEAR A
                PUTD,          ( PUSH D ONTO THE DATA STACK
                RESTORE-REGS,
                RTS,

END-CODE

```

```

( *****
( XIRQ-SERVICE - INTERRUPT ROUTINE FOR EXTERNAL INT LINE, I.E. 5380

```

```

(
( RUNS THROUGH A COMPLETE TARGET RECEIVE.  PLACES DATA INTO THE
( SCSI BUFFER, IN THE FOLLOWING ORDER.
(
(     SCSI_BUFFER + 0      INITIATOR ID
(     SCSI_BUFFER + 1      CMD BYTE 1
(     SCSI_BUFFER + 2      CMD BYTE 2
(     SCSI_BUFFER + 3      CMD BYTE 3
(     SCSI_BUFFER + 4      CMD BYTE 4
(     SCSI_BUFFER + 5      CMD BYTE 5
(     SCSI_BUFFER + 6      CMD BYTE 6
(     SCSI_BUFFER + 7      DATA BYTE 1
(
( *****
CODE-SUB XIRQ-SERVICE
    SCSI_ADO #      LDX,          ( SET X FOR SCSI POINTER )
    ,X 10 05        .SET.        ( SEE IF 5380 INTERRUPT BIT SET )
IF,                ( IF 5380 INTERRUPT CAUSE, THEN CONTINUE )
    ,X 80 05        .SET.        ( SEE IF EOP BIT IS SET )
IF,                ( IF EOP CAUSED INTERRUPT CONTINUE HERE )
    ,X 7            LDAA,         ( RESET THE INTERRUPT
    RTI,            ( RETURN FOR CONTINUED DMA SERVICE )

ELSE,              ( CONTINUE IF NOT EOP INTERRUPT )
    ,X 02 04        .CLR.        ( SEE IF SELECT LINE INACTIVE
IF,                ( IF NOT A SELECTION INTERRUPT THEN HERE )

( **** ADD CODE FOR OTHER 5380 INTERRUPTS IF NEEDED LATER **** )

ELSE,              ( IF A SELECTION INTERRUPT THEN HERE )
    SCSI_BUFF_FULL? LDAA,        ( SEE IF SCSI_BUFFER IS STILL FULL
    .NE.

IF,
    01 #           LDAA,         ( FLAG VALUE FOR SCSI_BUFF_OVFLW
    SCSI_BUFF_OVFLW STAA,        ( SET OVERFLOW FLAG
THEN,
    ,X 0           LDAA,         ( GET THE INITIATOR ID FROM BUS
    MON-SCSI-ID #  LDAB,         ( GET OUR ID FOR
    SBA,           ( SUBTRACT OUR ID FROM BUS VALUE
    SCSI_BUFFER    STAA,         ( STORE ID TO BUFFER
    08 #           LDAA,         ( LOAD A WITH ASSERT BSY VALUE
    ,X 1           STAA,         ( ASSERT BSY ON THE SCSI BUS
    06 #           LDAA,         ( SET LOOP CONTROL VALUE
BEGIN,
    PSHA,          ( ***FIX -- SAVE A FOR LOOP CONTROL
    CMD/STAT-PHASE # LDD,        ( SET D WITH CMD_PHASE VALUE
    PUTD,          ( SAVE 2 TO DATA STACK
    ' T_RECEIVE @  JSR,          ( JUMP TO SUBROUTINE FOR RECEIVE
    PULA,          ( ***FIX -- RESTORE A
    DECA,          ( DECREMENT A VALUE
    .EQ.           ( IS A 0 YET?
UNTIL,            ( LOOP UNTIL A IS 0.
    DATA-PHASE #  LDD,          ( SET DATA PHASE
    PUTD,          ( SAVE 0 TO STACK FOR DATA PHASE
    ' T_RECEIVE @  JSR,          ( DO DATA PHASE.

```

```

                                ( ***SET STATUS TO SEND TO PS/2
PORTA          LDAB,          ( SET B WITH PORTA
03 #          ANDB,          ( MASK OFF DMA NEED AND FIFO OVERFLOW ONLY
01 #          EORB,          ( INVERT THE DMA NEED BIT
PORTA          LDAA,          ( SET A WITH PORT D FOR DMA NEED BIT
FF #          EORA,          ( INVERT ALL BITS
                                ( SHIFT RESET BIT TO BIT 2
04 #          ANDA,          ( MASK OFF RESET BIT
                                ( ADD A AND B AND PUT INTO A
                                ABA,
PORTD          LDAB,          ( SET B WITH PORTD VALUE
18 #          ANDB,          ( MASK OFF ENABLE LINES
                                ABA,
                                ( ADD ENABLE LINES TO CURRENT STATUS
                                TAB,
                                ( PLACE STATUS VALUE INTO B
                                CLRA,
                                ( CLEAR A
                                PUTD,
                                ( PUT STATUS VALUE ON THE STACK
CMD/STAT-PHASE # LDD,          ( SET VALUE FOR STAT PHASE
                                PUTD,
' T_SEND @     JSR,          ( ***SEND STATUS DATA
                                CLRA,
                                ( CLEAR A
,X 0           STAA,          ( CLEAR DATA REGISTER
,X 1           STAA,          ( CLEAR INITIATOR COMMAND REG.
,X 3           STAA,          ( CLEAR TARGET CMD REG.
,X 7           LDAA,          ( RESET THE INTERRUPT
07 #           LDAA,          ( SET LOOP COUNTER VALUE
SCSI_BUFFER 7 + # LDX,          ( SET DESTINATION POINTER
BEGIN,
                                PSHA,
                                ( SAVE LOOP COUNTER VALUE
                                POPD,
                                ( GET SCSI CMD OR DATA VALUE
,X 0           STAB,          ( SAVE SCSI DATA VALUE
                                DEX,
                                ( DEC DESTINATION POINTER
                                PULA,
                                ( RESTORE LOOP COUNTER VALUE
                                DECA,
                                ( DECREMENT VALUE
                                .EQ.
                                ( SEE IF LOOP COUNTER IS 0
UNTIL,
01 #           LDAA,          ( SET A FOR SCSI_BUFFER_FULL FLAG
SCSI_BUFF_FULL? STAA,          ( INDICATE DATA IN SCSI_BUFFER
                                RTI,
                                ( THEN FOR SELECTION CODE CHOICE )
THEN,
                                ( THEN FOR EOP INTERRUPT CAUSE )
THEN,
                                ( THEN FOR NOT 5380 INTERRUPT )

( ***** ADD CODE HERE FOR OTHER INTERRUPT CAUSES IF NEEDED ***** )

END-CODE

( *****
( RTIINT - INTERRUPT ROUTINE FOR REAL TIME INTERRUPT
(
( SIMPLY CHECKS THE 4.1COUNT VARIABLE AND IF IT IS NOT
( 0 THEN IT DECREMENTS IT.
(
( *****

CODE-SUB RTIINT
4.1COUNT     LDX,          ( GET 4.1COUNT VALUE
.NE.          ( SEE IF IT IS ZERO

```

```

IF,
    DEX,          ( IF SO THEN DECREMENT X      )
    4.1COUNT    STX,          ( STORE NEW VALUE OF 4.1COUNT )
THEN,
    40 #         LDAA,         ( GET VALUE TO RESET RTI FLAG )
    TFLG2        STAA,         ( RESET THE RTI FLAG          )
    RTI,
END-CODE

```

```

( *****
( DMA_SERVICE - PERFORMS ALL OPS NEEDED FOR DMA SEND OF FIFO DATA
(
( NO STACK OPERATIONS NEEDED. SETS INITIATOR MODE AND CONTACTS
( PS/2. THEN SENDS 2 COMMAND BYTES, SETS DATA PHASE AND DMA
( MODE ON THE 5380 AND TELLS THE DMA CONTROLLER TO GO. ENTERS
( A WAIT STATE WHILE THE DMA
( CONTROLLER DOES ITS THING. THE DMA CONTROLLER ISSUES AN
( EOP INTERRUPT, HANDLED BY THE IRQ-SERVICE ROUTINE, AND RETURNS
( HERE. THIS ROUTINE THEN SETS THE 5380 TO TARGET MODE AND
( ENTERS BUS FREE PHASE, ENDING THE TRANSFER.
(
( *****

```

```

CODE-SUB DMA_SERVICE ( Tell DMA Controller to Go, ASSEMBLER FOR SPEED )

```

```

    SAVE-REGS, ( SAVE CALLER REGISTERS
    SCSI_ADO # LDX, ( SET POINTER TO SCSI REGISTERS
    00 #       LDAA, ( LOAD A WITH INIT MODE VALUE
    ,X 2       STAA, ( SET INITIATOR MODE
    CLRA,      ( CLEAR A TO RESET TARGET INTERRUPT
    ,X 4       STAA, ( RESETS THE TARGET INTERRUPT
    PS/2-ID #  LDAB, ( SET D TO TARGET ID
    PUTD,      ( PUT D ON FORTH DATA STACK
    ' ARB/SELECT @ JSR, ( DO ARBITRATION AND SELECTION
    POPD,      ( LOAD D WITH ERROR CODE FROM ARB/SELECT
    0 #        CPD, ( SEE IF D IS ZERO
    .EQ.       ( DID ERROR OCCUR?, 1=ERROR 0 = NO ERROR )
IF,
    BEGIN,
    ,X 4       LDAA, ( GET CURRENT SCSI CONTROL REGISTER
    1C #       ANDA, ( PARSE OUT PHASE INFORMATION
    08 #       CMPA, ( SEE IF IT IS COMMAND PHASE
    .EQ.       ( IF YES THEN CONTINUE
UNTIL,
    PUTD,      ( RETURN A GOOD ERROR CODE
    DUMP-DATA # LDAB, ( SET B WITH FIRST COMMAND BYTE
    CLRA,      ( CLEAR A TO PUT D ON FORTH DATA STACK )
    PUTD,      ( SET DATA TO FORTH DATA STACK
    CMD/STAT-PHASE # LDAB, ( SET B WITH COMMAND PHASE INFO
    PUTD,      ( SET PHASE INFO ON FORTH DATA STACK
    ' I_SEND @ JSR, ( SEND FIRST COMMAND TO PS/2
    CLRB,      ( CLEAR B FOR SECOND COMMAND BYTE
    PUTD,      ( SET SECOND COMMAND BYTE TO FORTH DATA STACK
    CMD/STAT-PHASE # LDAB, ( SET B WITH PHASE INFORMATION )
    PUTD,      ( PUT PHASE INFO ON FORTH DATA STACK
    ' I_SEND @ JSR, ( SEND SECOND COMMAND TO PS/2
    00 #       LDAA, ( SET A TO DATA OUT PHASE VALUE

```

```

,X 3      STAA,      ( SET TARGET CMD REG TO DATA OUT PHASE )
0A #      LDAA,      ( SET A FOR DMA, AND EODMA INTERRUPT )
,X 2      STAA,      ( SET THE MODE REGISTER WITH A VALUE )
,X 5      STAA,      ( WRITE TO START DMA SEND REGISTER
PORTD     LDAA,      ( GET PRESENT PORTD VALUE
04 #      ORAA,      ( SETS THE DMA_START SIGNAL
PORTD     STAA,      ( START THE DMA CYCLE
          SEI,        ( BLOCK ALL MASKABLE INTERRUPTS
          WAI,        ( WAIT UNTIL INTERRUPTED FOR DMA TRANSFER
BEGIN,    ( *** CHECKS TO SEE IF LAST BYTE SENT
,X 3      LDAA,      ( GETS THE TARGET CMD REG.
80 #      ANDA,      ( PARSE OUT "LAST BYTE SENT" BIT
80 #      CMPA,      ( SEE IF = TO 1
          .EQ.        ( ***IF LAST BYTE SENT THEN CONTINUE
UNTIL,
          CLI,        ( ALLOW ALL ENABLED INTERRUPTS
          CLRA,      ( SET A TO INITIATOR MODE
,X 2      STAA,      ( RESET THE DMA MODE BIT
,X 3      STAA,      ( SET TAR CMD REG. TO INITIAL PHASE VALUES
,X 1      STAA,      ( DEASSERT BSY AND DATA BUS
40 #      LDAA,      ( SET A TO VALUE FOR TARGET MODE
,X 2      STAA,      ( SET TARGET MODE )
MON-SCSI-ID # LDAA,  ( GET OUR ID VALUE
,X 4      STAA,      ( SET OUR ID, ENABLING SEL INT.
PORTD     LDAA,      ( GET PORT D VALUE
FB #      ANDA,      ( SET A TO /DMA_START SIGNAL VALUE
PORTD     STAA,      ( RESET THE DMA_START SIGNAL
          RESTORE-REGS, ( GET X,B,A FROM 6811 STACK
          RTS,        ( RETURN TO MAIN LOOP
THEN,
          PUTD,      ( PUT ERROR CODE BACK ON STACK
40 #      LDAA,      ( SET A TO VALUE FOR TARGET MODE
,X 2      STAA,      ( SET TARGET MODE )
MON-SCSI-ID # LDAA,  ( GET OUR ID VALUE
,X 4      STAA,      ( SET OUR ID FOR SELECTION INTERRUPT
          CLRA,      ( SET A TO PHASE VALUES
,X 3      STAA,      ( SET TAR CMD REG. TO INITIAL PHASE VALUES
          RESTORE-REGS, ( GET X,B,A FROM 6811 STACK
          RTS,        ( RETURN
END-CODE

```

```

( *****
( 4MS-RTI-SETUP - SETS UP THE 4MS RTI INTERRUPT
(
( THIS SUBROUTINE SIMPLY ENABLES THE RTI INTERRUPT
( AND SETS THE INTERRUPT RATE TO 4.1MS INTERRUPT
( *****

```

CODE-SUB 4MS-RTI-SETUP

```

PACTL     LDAA,      ( GET CURRENT VALUE OF PACTL
FC #      ANDA,      ( RESET THE RTI RATE BITS TO 4.1MS
PACTL     STAA,      ( RESTORE THE PACTL VALUE
TMSK2     LDAA,      ( GET TIMER MASK FOR ENABLING INT.
40 #      ORAA,      ( SET BIT 6 OF TMSK2 TO ENABLE INT.
TMSK2     STAA,      ( REWRITE TMSK2
          RTS,

```


END-CODE

```

( *****
( LOAD-INTVEC - SETS UP ALL INTERRUPT VECTORS, BUT DOES NOT
(           ENABLE THE INTERRUPTS.  A PROGRAM MUST USE
(           INTON TO DO THIS.  THIS ROUTINE USES ALL
(           FORTH AND PLACES THE INTERRUPT VECTORS IN
(           THEN APPROPRIATE EEPROM LOCATIONS.
(
( *****

: LOAD-INTVEC
                                7E B7E6 EEC!    ( LOADS THE BRA INSTUCTION
[ ' RTIINT @ >< FF AND ] LITERAL B7E7 EEC!    ( LOADS THE HI BYTE OF ADDRESS
[ ' RTIINT @      FF AND ] LITERAL B7E8 EEC!    ( LOADS THE LO BYTE OF ADDRESS

                                7E B7EC EEC!    ( LOADS THE BRA INSTUCTION
[ ' XIRQ-SERVICE @ >< FF AND ] LITERAL B7ED EEC!    ( LOADS HI BYTE OF ADDR
[ ' XIRQ-SERVICE @      FF AND ] LITERAL B7EE EEC!    ( LOADS LO BYTE OF ADDR
;

( Main Loop Functions Definitions )

( *****
(
( RESET_HARDWARE - ROUTINE TO ACTIVATE THE RESET LINE TO
(           THE UTUCAM HARDWARE.  THIS CLEARS EVERY
(           THING AND THEN EXITS.
(
( *****

: RESET_HARDWARE PORTA C@ F7 AND PORTA C! ;

( *****
( INIT5380 - INITIALIZES THE 5380 SCSI INTERFACE CHIP.
(
( CLEARS INITIATOR COMMAND REGISTER TO 00
( CLEARS THE TARGET COMMAND REGISTER
( RESETS ANY ERRORS OR INTERRUPTS PENDING
( CALLS ROUTINE TO SET TARGET MODE AND SET OUR ID.
(
( *****

: INIT5380    00 SCSI_ADO 1 + C!    ( CLEARS INITIATOR COMMAND REG.
              00 SCSI_ADO 3 + C!    ( CLEARS TARGET COMMAND REG.
              SCSI_ADO 7 + C@ DROP  ( RESETS ANY ERRORS/INTERRUPTS

              SET_TARGET_MODE      ( SETS ID AND TARGET BIT
;

( *****
( PRINTHEADER - PRINTS HEADER FOR THE MAIN LOOP
( USED IN THE IZATION ROUTINE
( *****

: PRINTHEADER CR

```

```

." -----" CR
." LOGIC BOX FIRMWARE" CR
." BY KEN BAKER" CR
." VERSION 1.06" CR
." MAR. 12, 1991" CR
." -----" CR
;

( *****
( INITIALIZE - THIS ROUTINE HANDLES ALL INITIALIZATION FOR
( THE MAIN LOOP.
(
( INITIALIZES FLAG VARIABLES
( INITIALIZES HARDWARE PORTS
( PUTS THE HARDWARE INTO RESET MODE
( SETS UP THE INTERRUPT VECTORS
( INITIALIZES THE SCSI BUS
( TURNS ON ALL INTERRUPTS BY THE I-BIT
(
( *****

( Initialize the SCSI, DMA, and Hardware)

: INITIALIZE

    INTOFF                ( DISABLE INTERRUPTS

    PRINTHEADER

    0 SCSI_BUFF_FULL? C!   ( Initializes SCSI_BUFFER_FULL )
    0 SCSI_BUFF_OVFLW C!   ( INITIALIZED OVERFLOW FLAG )
    0 4.1COUNT !          ( SET 4.1COUNT TO 0 )
    0 STATUS C!            ( CLEAR THE STATUS BYTE
    0 SCRATCH !            ( CLEARS THE SCRATCH

    8 0 DO 0 SCSI_BUFFER I + C! LOOP ( CLEARS SCSI BUFFER

    00 PORTD C!             ( SET INITIAL PORT D VALUE )
    1C DDRD C!              ( SETS DATA DIRECTION FOR PORT D )
    00 PORTA C!             ( SETS INITIAL VALUE FOR PORT A )

    LOAD-INTVEC             ( INITIALIZES INTERRUPT VECTORS )

    4MS-RTI-SETUP           ( ALLOWS 4MS-RTI SETUP )

    PACTL C@ 80 OR PACTL C! ( *** TEST CODE REMOVE IN FINAL ***

    INIT5380                ( INITIALIZE THE SCSI INTERFACE )

    INTON                   ( TURNS INTERRUPTS ON )
;

( *****
( PROCESS_SCSI - HANDLES HIGH SCSI COMMANDS FROM PS/2
(

```

```

( THE HIGH LEVEL COMMANDS ARE:
(
( SET_CONTROL_VALUE - DETERMINES WHETHER TO CAPTURE LEVELS, WAITS.
( RESET_MONITOR - RESETS THE MONITOR HARDWARE, THEN RESETS 6811.
( START_MONITOR - ENABLES MONITOR DATA CAPTURING.
( STOP_MONITOR - RESETS THE MONITOR HARDWARE.
( *****

: PROCESS_SCSI ( Processes any SCSI commands )

    SCSI_BUFFER 1 + C@ 0 = IF          ( PROCESSING FOR SET_CONTROL )
    SCSI_BUFFER 7 + C@ 8 * PORTD C@ E3 AND OR PORTD C!
    ." SET_CONTROL " CR ( THIS LINE FOR TESTING ONLY )
    THEN

    SCSI_BUFFER 1 + C@ 01 = IF          ( PROCESSING FOR RESET_MONITOR )
    RESET_HARDWARE
    COLD ( MUST BE SETUP FOR REAL FIRMWARE, NOT RAM TEST )
    ." RESETTING MONITOR " CR          ( THIS LINE FOR TESTING ONLY )
    THEN

    SCSI_BUFFER 1 + C@ 02 = IF          ( PROCESSING FOR START_MONITOR )
    PORTA C@ 08 OR PORTA C!            ( UNDOES THE RESET LINE )
    ." STARTING MONITOR " CR          ( THIS LINE FOR TESTING ONLY )
    THEN

    SCSI_BUFFER 1 + C@ 03 = IF          ( PROCESSING FOR STOP_MONITOR )
    RESET_HARDWARE
    ." STOPPING MONITOR " CR ( THIS LINE FOR TESTING ONLY )
    THEN

    0 SCSI_BUFF_FULL? C!
    ;

CODE-SUB DMA_READY?
    CLRB, ( SET B WITH DEFAULT VALUE )
    PORTA LDAA, ( GET CURRENT PORTD VALUE )
    01 # ANDA, ( PARSE OUT THE DMA_NEEDED LINE )
    .EQ. ( IF ZERO THEN RETURN A ONE )
IF,
    01 # LDAB, ( RETURN ONE )
THEN,
    CLRA, ( CLEAR A FOR STACK LOAD OF D )
    PUTD, ( LOAD D ONTO FORTH DATA STACK )
    RTS,
END-CODE

: ?KEY ?TERMINAL IF KEY ELSE 0 THEN ;

: EXIT-KEY?
    ?KEY ( GETS WHETHER KEY WAS HIT AND IF SO GETS VALUE )
    OB =
;

: DMA_ERROR ( ERROR ROUTINE FOR DMA ATTEMPT )

```

```
CR ." DMA_ERROR " DUP 01 = IF DMA_SERVICE DUP 0 = IF DROP ELSE
  RESET_HARDWARE THEN THEN ;

: MAIN INITIALIZE
  BEGIN
    SCSI_BUFF_FULL? C@ IF PROCESS_SCSI THEN
      DMA_READY? IF DMA_SERVICE DROP THEN ( DROPPING ERROR CODE )
      EXIT-KEY? IF CR ." KEY EXIT" CR ABORT THEN
    O UNTIL ;

D000 AUTOSTART MAIN ( SETS AUTOSTART FOR MAIN ROUTINE )
```

APPENDIX D

PAL SPECIFICATION LISTINGS

This appendix gives the listing of the PAL specifications used to implement the various state machines and data converters on the Monitor. These PAL listings are given in the PALASM4 language available from Advanced Micro Devices, Inc. The order of the PAL listings are given below.

LEVCAP06.PDS	Priority Level Change Capture PAL
WAITCAP3.PDS	WAIT State Entry and Exit Capture PAL
WRTCNT04.PDS	Write Controller PAL
COMBWC02.PDS	Comment Byte Write Converter PAL
CMDBWC02.PDS	Command Byte Write Converter PAL
DMACNT06.PDS	DMA Controller PAL
SCSI_CH2.PDS	68HC11 to SCSI Converter PAL

•

LEVCAP06.PDS

```

;*****
;*
;*      LVLCAPO6.PDS
;*
;*      This file specifies the palasm assembly
;*      for the level capture pal.  the pal
;*      is a 16R6 and is used in a sync. clocked mode
;*      design.
;*
;*      BASIC OPERATIONAL DESC.
;*
;*      This pal implements three synchronous state machines
;*      used in the capturing of level information for the
;*      UTUCAM project.  The three state machines are the
;*      CAPTURE_LPL machine, CAPTURE_CPL machine and the
;*      LC machine.  CAPTURE_LPL and CAPTURE_CPL are used to
;*      control 2 external four bit registers, 74LS175's.
;*      The LC lines is used to indicate to the Write Ckt, and
;*      to the Write Controller that a level change has taken
;*      place and that the LPL and CPL data has been taken
;*
;*      The chip has 7 inputs, six state variables
;*      and three outputs (corrospound to the state variables
;*
;*      The seven inputs are:
;*          clock
;*          /IFWT
;*          /IRPT
;*          /RSEL
;*          /WRT
;*          EN_LEVEL_CAP
;*          /RESET_LEVEL
;*
;*      The six state variables are
;*          LPLS1
;*          LPLS0
;*          CPLS1
;*          CPLS0
;*          /LCS1
;*          /LCS0
;*
;*      The ouputs are
;*          CAPTURE_LPL      =      LPLS0
;*          CAPTURE_CPL      =      CPLS0
;*          /LC              =      LCS0
;*
;*      The chip also contains logic to decode two
;*      signals called LPL_AVAIL and CPL_AVAIL, which
;*      indicate when LPL and CPL are available on the
;*      address bus.  These signals are on the two
;*      non-registered outputs.
;*
;*      FOR VERSION 3 OF THIS CIRCUIT, A SIGNAL CALLED
;*      Change_en was added.  This signal is needed because
;*      the circuit needs to freeze the information in time
;*      until that information has been written into the

```

```

;*      FIFO memory, then changes can be made to the information
;*      The change_en signal only affects the registered state
;*      machines, not the combo outputs.
;*
;*
;*      FOR VERSION 4 OF THIS CIRCUIT, THE OVERALL RESET
;*      signal was added to the circuit. The reset was
;*      added to pin 9, filling the PAL inputs and outputs.
;*      This reset is the overall system reset and is active
;*      low. This is the only change to version 4, from
;*      version 3. The simulation section was added to test
;*      the actions of the reset circuit
;*
;*      FOR VERSION 5 OF THIS CIRCUIT, THE STATE SIGNALS
;*      and the outputs are made active low. I simply
;*      changes the polarity on the input and output signals
;*      to reflect the change in polarity.
;*
;*****

```

```

TITLE          LEVEL CAPTURE PAL
PATTERN        EVENT CAPTURE CIRCUITS
REVISION       06
AUTHOR         KEN BAKER
COMPANY        UNIVERSITY OF TENNESSEE
DATE           JUNE 19,1990

```

```

CHIP           LEVEL_CAP_PAL           PAL16R6

```

```

;PINOUT      1      2      3      4      5      6      7

```

```

          CLOCK /IFWT /IRPT /RSEL /WRT  EN_LEVEL_CAP /LEVEL_RESET

```

```

;PINOUT      8      9      10

```

```

          CHANGE_EN /RESET GND

```

```

;PINOUT 11      12      13      14      15      16      17      18      19      20

```

```

          /OE CPL_AVAIL /LCS1 /LCS0 CPLS1 CPLS0 LPLS1 LPLS0 LPL_AVAIL VCC

```

```

EQUATIONS

```

```

;ENCODER EQUATIONS

```

```

LPL_AVAIL = IFWT*WRT*IRPT*RSEL
LPL_AVAIL.TRST = VCC
CPL_AVAIL = IFWT*/WRT*IRPT*RSEL
CPL_AVAIL.TRST = VCC

```

```

; STATE EQUATIONS

```

```

LPLS1 := LPL_AVAIL*CHANGE_EN*/RESET
      + LPLS0*CHANGE_EN*/RESET

```



```

+ /CHANGE_EN*LPLS1*/RESET

LPLS0 := LPL_AVAIL*/LPLS1*CHANGE_EN*/RESET
+ /CHANGE_EN*LPLS0*/RESET

CPLS1 := CPL_AVAIL*CHANGE_EN*/RESET
+ CPLS0*CHANGE_EN*/RESET
+ /CHANGE_EN*CPLS1*/RESET

CPLS0 := CPL_AVAIL*/CPLS1*CHANGE_EN*/RESET
+ /CHANGE_EN*CPLS0*/RESET

LCS1 := CPL_AVAIL*/LEVEL_RESET*EN_LEVEL_CAP*CHANGE_EN*/RESET
+ CPL_AVAIL*LCS1*EN_LEVEL_CAP*CHANGE_EN*/RESET
+ LCS0*EN_LEVEL_CAP*CHANGE_EN*/RESET
+ /CHANGE_EN*LCS1*/RESET

LCS0 := /LEVEL_RESET*LCS1*LCS0*EN_LEVEL_CAP*CHANGE_EN*/RESET
+ CPL_AVAIL*/LEVEL_RESET*/LCS1*EN_LEVEL_CAP*CHANGE_EN*/RESET
+ /CHANGE_EN*LCS0*/LEVEL_RESET*/RESET

SIMULATION
;***      SETTING THE MACHINES IN KNOWN STATE

SETF /EN_LEVEL_CAP /LEVEL_RESET OE CHANGE_EN /RESET
SETF /IFWT /WRT /IRPT /RSEL

CLOCKF                      ;SET EVERYTHING TO ZERO STATE
CLOCKF
CLOCKF
CLOCKF

TRACE_ON EN_LEVEL_CAP LEVEL_RESET CHANGE_EN RESET CPL_AVAIL CLOCK LCS1 LCS0
SETF EN_LEVEL_CAP RESET

CLOCKF
CLOCKF

SETF /RESET IFWT IRPT RSEL

CLOCKF
CLOCKF
CLOCKF

SETF /IFWT

CLOCKF
CLOCKF
CLOCKF

SETF LEVEL_RESET

CLOCKF
CLOCKF

TRACE_OFF

```

WAITCAP3.PDS

```

;*****
;*
;* WAITCAP3.PDS - PALASM2 input file describing the
;* wait capture pal for the utucam project. This pal
;* will watch the wait signal from the redrive card and
;* log wait entry and wait exit.
;*
;* Inputs and outputs
;*
;* The inputs to the wait capture pal are given below.
;*
;* /WAIT - the wait signal from the redrive card
;* RWEX - resets the wait exit line, from write controller
;* RWEN - resets the wait entry line, from wr controller
;* CHANGE_EN - allows the inputs to affect the state of machine
;* /RESET - resets the machine to initial state
;*
;* The outputs of the wait capture pal are:
;*
;* WAIT_ENTRY - indicates that a wait entry has occurred
;* WAIT_EXIT - indicates that a wait exit has occurred.
;*
;* Basic Function of circuit
;*
;* The circuit operation is ... The CHANGE_EN signal is a system
;* change enable, and must be high for changes at the inputs to
;* change the state of the machine. The /RESET signal must also be
;* high for operation of the circuit.
;*
;* When the circuit has been properly reset, it is in state 1, and will
;* remain there until it sees wait go active. It will then output a
;* WAIT_ENTRY. It will hold the WAIT_ENTRY signal active until a
;* RESET_WAIT_ENTRY (RWEN) signal is received. When the wait signal goes
;* inactive, the WAIT_EXIT signal is activated. The WAIT_EXIT
;* signal will stay active until a RESET_WAIT_EXIT (RWEX)
;* signal is received. This is the basic operation of the
;* circuit.
;*
;* General Notes
;*
;* This circuit is designed to allow for the fact that a
;* wait entry and a wait exit can occur in the same microsecond.
;* This could occur as follows. Assume that a wait
;* state is entered, and immediately following the wait
;* state, an interrupt occurs, causing a wait exit. I
;* have no specific timing for this event, and the inputs
;* must stay stable for the length of time of a write
;* cycle (as timer overflow writes have the highest priority
;* but the circuit will catch wait entry and exit in the
;* same usecond.
;*
;*
;* FOR VERSION 1.1 of this specifications, the EN_WAIT_CAP line
;* was added to the equations. This line was on the pin
;* list but was not used in the first revision.
;* This line is necessary for the circuit

```

```

;*      so that the 68HC11 microprocessor can specify whether
;*      the waits are supposed to be captured or not.  Without
;*      this line, the waits would always be captured.
;*
;*      FOR VERSION 2 OF THIS PAL SPECIFICATIONS, two minterms were
;*      added to ws2 and ws1 equations to make state 5 more robust.
;*      These changes were made on 9/10/90
;*
;*
;*      FOR VERSION 3 OF THIS PAL, THE CHANGE_EN BUG WAS FIXED.
;*      Basically, the pal must be allowed to make certain state
;*      transitions even when change_en is low.
;*
;*****

```

TITLE	WAIT_CAPTURE_PAL
PATTERN	EVENT CAPTURE CIRCUITS
REVISION	03
AUTHOR	KEN BAKER
COMPANY	UNIVERSITY OF TENNESSEE
DATE	SEPT 15, 1990

CHIP WAIT CAP PAL PAL16R8

```
:PINOUT      1      2          3          4      5      6          7
```

CLOCK /WAIT EN WAIT CAP /RESET /RWEN /RWEX CHANGE EN

```

:PINOUT      8   9   10

```

NC NC GND

```
:PINOUT 11 12 13 14 15 16 17 18 19 20
```

```

/OE NC NC NC NC NC /WS2 /WS1 /WS0 VCC

```

EQUATIONS

```
;EQUATIONS FOR WAIT STATE MACHINE ARE NOT REDUCED IN THE LEAST
;These equations came straight from the state table, and
;are not reduced at all.  I am going to let PALASM2 reduce
;them for me.
```

```
WS0 := /WS2*/WS1*WS0*/WAIT*/RWEX*/RWEN*EN WAIT CAP*CHANGE EN*/RESET
```

```
+ /WS2*WS1*WS0*/WAIT*/RWEX*/RWEN*EN WAIT CAP*CHANGE EN*/RESET
```

```
+ /WS2*/WS1*/WS0*WAIT*/RWEX*/RWEN*EN WAIT CAP*CHANGE EN*/RESET
```

```
+ /WS2*/WS1*WS0*WAIT*/RWEX*/RWEN*EN WAIT CAP*CHANGE EN*/RESET
```

```
+ /CHANGE_EN*/RWEN*WSO*/RESET*EN_WAIT_CAP      ;FEEDBACK EQUATION, WITH
```

;CHANGE EN FIX

```

WS1 := /WS2*/WS1*WS0*/WAIT*/RWEX*/RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + /WS2*WS1*WS0*/WAIT*/RWEX*/RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + WS2*/WS1*/WS0*/WAIT*/RWEX*/RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + WS2*WS1*/WS0*/WAIT*/RWEX*/RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + /WS2*/WS1*WS0*/WAIT*/RWEX*RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + /WS2*WS1*WS0*/WAIT*/RWEX*RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + WS2*/WS1*/WS0*/WAIT*/RWEX*RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + WS2*WS1*/WS0*/WAIT*/RWEX*RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + WS2*WS1*/WS0*WAIT*/RWEX*/RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + /CHANGE_EN*/RWEX*WS1*/RESET*EN_WAIT_CAP
      + /CHANGE_EN*RWEX*/WS2*WS1*WS0*/RESET*EN_WAIT_CAP

```

```

WS2 := WS0*/RWEX*RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + WS2*/RWEX*EN_WAIT_CAP*CHANGE_EN*/RESET
      + WS2*WS1*/WS0*WAIT*/RWEX*/RWEN*EN_WAIT_CAP*CHANGE_EN*/RESET
      + /CHANGE_EN*/RWEN*/RWEX*WS2*/RESET*EN_WAIT_CAP      ;FEEDBACK EQUATION
      + /CHANGE_EN*/RWEN*RWEX*WS2*/WS1*/WS0*/RESET*EN_WAIT_CAP      ;FOR /RWEN*RWEX
      + /CHANGE_EN*RWEN*/RWEX*WS2*/WS0*/RESET*EN_WAIT_CAP      ;FOR RWEN*/RWEX EQ. 1
      + /CHANGE_EN*RWEN*/RWEX*/WS2*WS0*/RESET*EN_WAIT_CAP      ;FOR RWEN*/RWEX EQ. 2

```

SIMULATION

```

;SIMULATION SECTION FOR THE WAIT CAPTURE PAL

```

```

;
;*** THIS SECTION WILL FIRST PUT THE PAL INTO A KNOWN STATE
;*** THEN WILL TEST NORMAL OPERATION OF THE CIRCUIT
;*** THEN WILL TEST RESET CAPABILITIES OF THE CIRCUIT

```

```

;* Putting the PAL into a known state

```

```

TRACE_ON /WAIT EN_WAIT_CAP /RESET RWEX RWEN CHANGE_EN WS2 WS1 WS0 CLOCK

```

```

SETF /WAIT EN_WAIT_CAP /RESET /RWEX /RWEN CHANGE_EN OE EN_WAIT_CAP

```

```

CLOCKF

```

```

CLOCKF

```

```

SETF RESET

```

CLOCKF
CLOCKF

SETF /RESET

```

;*****
;***
;***      Testing a normal wait transfer with the following situation
;***
;***      a reset has already occurred, to put the machine into
;***      state one. Wait will go active, causing a transition
;***      to state two, and an activation of the WAIT_ENTRY output.
;***      This is followed by activation of the
;***      RWEN (reset wait entry) line, causing a transition
;***      to state 4. WAIT will then go inactive, causing a
;***      transition to state 5, and activation of the WAIT_EXIT
;***      signal. This is followed by an active RWEX, sending the
;***      machine back to state one, and a deactivation of the
;***      WAIT_EXIT signal.
;***
;*****

```

```
SETF WAIT      ;active wait, trans. to state two
```

CLOCKF
CLOCKF

```
SETF RWEN          ;reset the wait log, causing trans. to state 4
```

CLOCKF
CLOCKF

```
SETF /RWEN      ;deactivate rwen no transition
```

CLOCKF

```
SETF /WAIT          ;deactivate wait, for wait_exit log
```

CLOCKF
CLOCKF

```
SETF RWEX          ;clear the wait exit log. to state one
```

CLOCKF
CLOCKF

```
SETF /RWEX          ;sends the machine to state one
```

```

;*****
;***
;***      Testing a normal wait transfer with the wait
;***      going active and inactive before RWEN or RWEX
;***      go active.
;***
;***      The sequence of events is, Wait goes active, sending
;***      the machine into state two, from state one. Then
;***      Wait goes inactive, causing a transtion to state 3.
;***      Then RWEN and RWEX are activated for resetting the
;***      respective events.
;***
;*****

SETF WAIT          ;active wait, trans. to state two

CLOCKF
CLOCKF

SETF /WAIT         ;causes trans. to state 3, active WAIT_ENTRY and WAIT_EXIT

CLOCKF
CLOCKF

SETF RWEN          ;deactivate rwen no transition

CLOCKF

SETF /RWEN         ;deactivate wait, for wait_exit log

CLOCKF
CLOCKF

SETF RWEX          ;clear the wait_exit log. to state one

CLOCKF
CLOCKF

SETF /RWEX         ;keeps the machine in state one
;*****
;***
;***      Testing a normal wait transfer with the wait
;***      going active, being reset by RWEN, and RWEN being active
;***      when WAIT goes inactive
;***
;***      The sequence of events is, Wait goes active, sending
;***      the machine into state two, from state one. Then
;***      RWEN is active, causing a transtion to state 4.
;***      RWEN is held active while WAIT goes inactive, to see
;***      what will happen. then RWEN will be deactivated, and
;***      RWEX will be activated and deactivated.
;***
;*****

SETF RESET         ;Puts the machine into state one

```

```

CLOCKF
CLOCKF

SETF /RESET

CLOCKF
CLOCKF

SETF WAIT                ;active wait, trans. to state two

CLOCKF
CLOCKF

SETF RWEN                ;Resets the WAIT_ENTRY output

CLOCKF
CLOCKF

SETF /WAIT              ;causes trans. to state 5, active WAIT_ENTRY and WAIT_EXIT

CLOCKF
CLOCKF

SETF /RWEN              ;deactivate rwen no transition

CLOCKF
CLOCKF

SETF RWEX                ;clear the wait_exit log. to state one

CLOCKF
CLOCKF

SETF /RWEX              ;keeps the machine in state one

;*****
;***
;***      Testing a normal wait transfer with the wait
;***      going active, being reset by RWEN, and RWEN being active
;***      when WAIT goes inactive. RWEN will also stay active
;***      while RWEX is active.
;***
;***      The sequence of events is, Wait goes active, sending
;***      the machine into state two, from state one. Then
;***      RWEN is active, causing a transtion to state 4.
;***      RWEN is held active while WAIT goes inactive, to see
;***      what will happen. then RWEX will be activated, and
;***      RWEN will be deactivated amd RWEX will be deactivated.
;***
;*****

SETF RESET              ;Puts the machine into state on

CLOCKF
CLOCKF

SETF /RESET

```


CLOCKF
CLOCKF

SETF WAIT ;active wait, trans. to state two

CLOCKF
CLOCKF

SETF RWEN ;Resets the WAIT_ENTRY output

CLOCKF
CLOCKF

SETF /WAIT ;causes trans. to state 5, active WAIT_ENTRY and WAIT_EXIT

CLOCKF
CLOCKF

SETF RWEX ;deactivate rwex no transition

CLOCKF
CLOCKF

SETF /RWEN ;deactivate RWEN.

CLOCKF
CLOCKF

SETF /RWEX ;keeps the machine in state one

SETF RESET

CLOCKF
CLOCKF

SETF /RESET

```

;*****
;***
;***      Testing a normal wait transfer with the following situation
;***
;***      a reset will occurred, to put the machine into
;***      state one. Wait will go active, causing a transition
;***      to state two, and an activation of the WAIT_ENTRY output.
;***      The wait signal will then go inactive causing a tran. to state
;***      three, and output of the WAIT_EXIT output.
;***      This is followed by activation of the
;***      RWEN (reset wait entry) line, causing a transition
;***      to state 4. This is followed by an active RWEX, sending the
;***      machine back to state one, and a deactivation of the
;***      WAIT_EXIT signal.
;***
;*****

```

SETF WAIT ;active wait, trans. to state two

```

CLOCKF
CLOCKF

SETF /WAIT          ;deactivate WAIT causing trans. to state 3

CLOCKF
CLOCKF

SETF RWEN           ;activate rwen, transition to state 5

CLOCKF

SETF /RWEN          ;deactivate RWEN

CLOCKF
CLOCKF

SETF RWEX           ;clear the wait_exit log. to state one

CLOCKF
CLOCKF

SETF /RWEX          ;sends the machine to state one
SETF RESET

CLOCKF
CLOCKF

SETF /RESET

;*****
;***
;***   Testing the use of the EN_WAIT_CAP line.
;***
;*****

SETF /EN_WAIT_CAP    ;disables the machine.

SETF WAIT            ;active wait, trans. to state two

CLOCKF
CLOCKF

SETF /WAIT          ;deactivate WAIT causing trans. to state 3

CLOCKF
CLOCKF

SETF RWEN           ;activate rwen, transition to state 5

CLOCKF

SETF /RWEN          ;deactivate RWEN

CLOCKF

```

```

CLOCKF

SETF RWEX                ;clear the wait_exit log. to state one

CLOCKF
CLOCKF

SETF /RWEX                ;sends the machine to state one

;*****
;testing of the change_en problem
;* TESTS 1-2-4-5 TRANSITION
;*****

SETF /WAIT EN_WAIT_CAP /RESET CHANGE_EN /RWEN /RWEX
SETF RESET
CLOCKF
SETF /RESET
CLOCKF
SETF WAIT
CLOCKF
CLOCKF
SETF /CHANGE_EN
CLOCKF
CLOCKF
SETF RWEN
CLOCKF
CLOCKF
SETF /RWEN
CLOCKF
CLOCKF
SETF CHANGE_EN
CLOCKF
CLOCKF
SETF /WAIT
CLOCKF
CLOCKF
SETF /CHANGE_EN
CLOCKF
CLOCKF
CLOCKF
SETF RWEX
CLOCKF
CLOCKF
SETF /RWEX
CLOCKF
CLOCKF

;*****
;testing of the change_en problem
;* TESTS 1-2-3-5 TRANSITION
;*****

SETF /WAIT EN_WAIT_CAP /RESET CHANGE_EN /RWEN /RWEX
SETF RESET
CLOCKF

```

```

SETF /RESET
CLOCKF
SETF WAIT
CLOCKF
CLOCKF
SETF /WAIT
CLOCKF
CLOCKF
SETF /CHANGE_EN
CLOCKF
CLOCKF
SETF RWEN
CLOCKF
CLOCKF
SETF /RWEN
CLOCKF
CLOCKF
SETF RWEX
CLOCKF
CLOCKF
SETF /RWEX
CLOCKF
CLOCKF
SETF CHANGE_EN
CLOCKF
CLOCKF

```

```

;*****
;* testing of the change_en problem
;* tests what happens is wait goes inactive while
;* the machine is in state2 and change_en is inactive
;*****

```

```

SETF /WAIT EN_WAIT_CAP /RESET CHANGE_EN /RWEN /RWEX
SETF RESET
CLOCKF
SETF /RESET
CLOCKF
SETF WAIT
CLOCKF
CLOCKF
SETF /CHANGE_EN
CLOCKF
CLOCKF
SETF /WAIT
CLOCKF
CLOCKF
SETF RWEN
CLOCKF
CLOCKF
SETF /RWEN
CLOCKF
CLOCKF
SETF CHANGE_EN
CLOCKF
CLOCKF

```

```

SETF /CHANGE_EN
CLOCKF
CLOCKF
SETF RWEX
CLOCKF
CLOCKF
SETF /RWEX
CLOCKF
CLOCKF

;*****
;* testing of the change_en problem
;* tests what happens is wait goes inactive while
;* the machine is in state2 and change_en is inactive
;*****

SETF /WAIT EN_WAIT_CAP /RESET CHANGE_EN /RWEN /RWEX
SETF RESET
CLOCKF
SETF /RESET
CLOCKF
SETF WAIT
CLOCKF
CLOCKF
SETF /CHANGE_EN
CLOCKF
CLOCKF
CLOCKF
SETF RWEN
CLOCKF
CLOCKF
SETF /RWEN
CLOCKF
CLOCKF
SETF /WAIT
CLOCKF
CLOCKF
SETF CHANGE_EN
CLOCKF
CLOCKF
CLOCKF
SETF /CHANGE_EN
CLOCKF
CLOCKF
CLOCKF
SETF RWEX
CLOCKF
CLOCKF
SETF /RWEX
CLOCKF
CLOCKF
;*****
;***      End of the Simulation
;*****
TRACE_OFF

```

WRTCNT04.PDS

```

;*****
;*
;* Wrtcnt04.PDS - PALASM2 input file describing the
;* write controller state machine. This machine is made
;* to interface to the write signal encoder, the reset demux,
;* the event register,
;* and the wait and level capture circuits.
;*
;* The inputs to the write controller state machine are:
;*
;*     CLOCK      - 16 MHz clock signal for state machine
;*     /RESET     - syncn. reset signal
;*     WR_EN      - indicator for start write cycle
;*     EI         - event indicator signal
;*
;* The outputs to the write controller are:
;*
;*     RESET_EVENT - signal to reset demux for current event reset
;*     W0123       - signal for FIFO's to write current data
;*     HOLD        - tells logging circuits to NOT change there data
;*     CLOCK_EVENT - loads the next value of event into the register
;*
;* Basic operation of the circuit are as follows:
;*
;* /RESET and CLOCK are system signals that control operation
;* of the circuit overall. /RESET will reset the machine to
;* state 1 anytime it occurs, and it is a synchronous reset.
;* CLOCK is the system clock running at 16 MHz.
;*
;* WR_EN is the 1MHz clock from the clocking circuit and is
;* used to know when to write the data for the microsecond.
;* When the state machine sees a transition from high to
;* low, then the start of a new write cycle is indicated
;*
;* EI is the event indicator signal from the write signal
;* encoder. It tells the controller whether an event has
;* taken place, and whether events still remain to be written
;*
;* To operate, the machine first looks for a high to low
;* transition on the WR_EN line, indicating the start of a
;* new write cycle. when this has occurred, the machine activates
;* the HOLD output, freezing changes from occurring the the
;* logging circuit. The machine then
;* checks the EI signal to see if any events are in the queue to
;* be written. If nothing exists, then the machine simply DEactivates
;* the HOLD signal allowing new changes to occur by
;* waiting for the next high to low transition on the WR_EN line.
;*
;* If events are logged in the queue for writing, the machine
;* will activate the W1023 line to write the converted value into
;* FIFO memory, through the write circuit, then will reset the event
;* through the RESET_EVENT signal. The state machine then repeats
;* this action until all pending events have been written to
;* the FIFO memory.
;*
;* This machine is designed to operate at a speed of 16 MHz, and

```

```

;*  at that speed, it will write three events in a one usec time
;*  frame.
;*
;*  Design by Ken Baker
;*  For the UTUCAM project
;*
;*  FOR VERSION 2 OF THIS PAL, THE WR_EN LINE IS
;*  taken to be an active high signal instead of the
;*  previous active low signal.  The change is made by simply
;*  changing the polarity on the pin specification list
;*
;*****
;*
;*  FOR VERSION 3 OF THIS PAL, TWO STATES HAVE BEEN ADDED
;*  to allow for proper resetting of the FIFO memory.  In the
;*  past, I have held the write line, w0123 low when resetting
;*  the FIFOs.  This machine simply adds two front end states
;*  to the old state machine.  Reference changes on 2/4/91.
;*  I also have changed the notation from Boolean to
;*  state description.
;*
;*****
;*
;*  FOR VERSION 4 OF THIS PAL, I HAVE REDESIGNED THE MACHINE
;*  so that the w0123 signal operated properly.  I have added
;*  an extra state to allow for correct operation.  This
;*  design is shown on 3/2 and 3/3/91.
;*
;*****

TITLE      WRT_CNTRL_PAL
PATTERN    EVENT CAPTURE CIRCUITS
REVISION   04
AUTHOR     KEN BAKER
COMPANY    UNIVERSITY OF TENNESSEE
DATE       March 3, 1991

CHIP        WRT_CNTRL_PAL          PAL16R8

;PINOUT    1    2    3        4    5    6    7    8    9    10

            CLOCK EI  WR_EN  /RESET  NC  NC  NC  NC NC GND

;PINOUT    11    12    13  14 15  16  17

            /OE  /HOLD /W0123 NC WC0 WC1 WC2

;PINOUT    18        19        20

            RESET_EVENT CLOCK_EVENT VCC

;----- START OF THE STATE SECTION -----

STATE
MEALY_MACHINE

```



```

START_UP := POWER_UP -> START
START_UP.OUTF := POWER_UP -> /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

;*****
; STATE ASSIGNMENT
;*****

START          = /WC2*/WC1*/WC0      ;STATE 1 IN THE NOTEBOOK
NO_EVENT       = /WC2*/WC1*/WC0      ;STATE 2 IN THE NOTEBOOK
EVENT_HOLD    = /WC2*WC1*/WC0      ;STATE 3 IN THE NOTEBOOK
EI_DECISION    = /WC2*WC1*/WC0      ;STATE 4 IN THE NOTEBOOK
WRITE_DELAY    = WC2*/WC1*/WC0      ;STATE 5 IN THE NOTEBOOK
RESET_DELAY    = WC2*/WC1*/WC0      ;STATE 6 IN THE NOTEBOOK
SPACER         = WC2*WC1*/WC0      ;STATE 7 IN THE NOTEBOOK
SPACER2        = WC2*WC1*/WC0      ;STATE 8 IN THE NOTEBOOK

;*****
; STATE TRANSITIONS
;*****

START          := CON2              -> NO_EVENT
               + CON3              -> EVENT_HOLD
               +->                  START

NO_EVENT       := CON2              -> NO_EVENT
               + CON3              -> NO_EVENT
               +->                  START

EVENT_HOLD     := RST               -> START
               +->                  EI_DECISION

EI_DECISION    := RST               -> START
               + CON1              -> WRITE_DELAY
               + CON3              -> WRITE_DELAY
               + CON0              -> NO_EVENT
               + CON2              -> NO_EVENT

WRITE_DELAY    := RST               -> START
               +->                  RESET_DELAY

RESET_DELAY    := RST               -> START
               +->                  SPACER

SPACER         := RST               -> START
               +->                  SPACER2

SPACER2        := RST               -> START
               +->                  EVENT_HOLD

;*****
; OUTPUT EQUATIONS
;*****

START.OUTF     := CON3              -> /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT
               +->                  /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

```

```

NO_EVENT.OUTF      := VCC          -> /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

EVENT_HOLD.OUTF    := /RST         -> /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT
                  +->              /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

EI_DECISION.OUTF   := CON1         -> W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT
                  + CON3          -> W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT
                  +->              /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

WRITE_DELAY.OUTF   := /RST         -> W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT
                  +->              /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

RESET_DELAY.OUTF   := /RST         -> /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT
                  +->              /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

SPACER.OUTF        := /RST         -> /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT
                  +->              /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

SPACER2.OUTF       := /RST         -> /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT
                  +->              /W0123*/HOLD*/CLOCK_EVENT*/RESET_EVENT

```

```

;*****
;*   CONDITION DEFINITION
;*****

```

CONDITIONS

```

RST = RESET
CON0 = /WR_EN*/EI*/RESET
CON1 = /WR_EN*/EI*/RESET
CON2 = WR_EN*/EI*/RESET
CON3 = WR_EN*/EI*/RESET

```

```

;*****
;*   END OF THE EQUATIONS SECTION
;*****

```

SIMULATION

```

;*****
;*   SIMULATION SECTION FOR THE WRITE CONTROLLER PAL
;*   setting the machine to a known state
;*****

```

```
TRACE_ON CLOCK /RESET EI WR_EN /HOLD CLOCK_EVENT RESET_EVENT /W0123
```

```

SETF RESET /EI WR_EN OE
CLOCKF
CLOCKF

```

```

;*****
;*   testing normal operation of the controller for
;*   no events pending
;*****

```

```

SETF /RESET

CLOCKF                ;STATE 1 TO 2

SETF /WR_EN

CLOCKF                ;STATE 2 TO 1
CLOCKF

SETF WR_EN

CLOCKF                ;STATE 1 TO 2
CLOCKF

;*****
;*   testing normal operation of the controller for
;*   one event write
;*****

SETF RESET  /WR_EN

CLOCKF                ;STATE 1

SETF /RESET

CLOCKF                ;to state 2

SETF EI

CLOCKF                ;stay in state 2

SETF WR_EN

CLOCKF                ; state 1
CLOCKF                ; state 3
CLOCKF                ; state 4
CLOCKF                ; state 5
CLOCKF                ; state 6
CLOCKF                ; state 7

SETF /EI

CLOCKF                ; state 3

SETF /WR_EN
CLOCKF                ; state 4

CLOCKF                ; state 2
CLOCKF                ; state 1
CLOCKF                ; state 1

;*****
;*   test a transfer with two events
;*****

SETF RESET                ; putting into a known state

```

```

CLOCKF

SETF /RESET

SETF EI /WR_EN

CLOCKF                ; state 1
CLOCKF                ; state 1

SETF WR_EN

CLOCKF                ; state 1 TO 2
CLOCKF                ; to state 3
CLOCKF                ; to state 4
CLOCKF                ; to state 5
CLOCKF                ; to state 6
CLOCKF                ; to state 7
CLOCKF                ; to state 3
CLOCKF                ; to state 4

SETF /WR_EN           ;INDICATES THE EIGHTH CLOCK CYCLE

CLOCKF                ; to state 5
CLOCKF                ; to state 6

SETF /EI

CLOCKF                ; to state 7
CLOCKF                ; to state 3
CLOCKF                ; to state 4
CLOCKF                ; to state 2
CLOCKF                ; to state 1
CLOCKF

;*****
;*   test a transfer with three events
;*****

SETF RESET            ; putting into a known state

CLOCKF                ; to state 1
CLOCKF

SETF /RESET /WR_EN EI

CLOCKF                ; to state 1
CLOCKF

SETF WR_EN

CLOCKF                ; to state 2
CLOCKF                ; to state 3
CLOCKF                ; to state 4
CLOCKF                ; to state 5
CLOCKF                ; to state 6

```

```
CLOCKF          ; to state 7
CLOCKF          ; to state 3
CLOCKF          ; to state 4

SETF /WR_EN      ;INDICATES THE EIGHTH CLOCK CYCLE

CLOCKF          ; to state 5
CLOCKF          ; to state 6
CLOCKF          ; to state 7
CLOCKF          ; to state 3
CLOCKF          ; to state 4
CLOCKF          ; to state 5
CLOCKF          ; to state 6

SETF /EI

CLOCKF          ; to state 7
CLOCKF          ; to state 3

SETF WR_EN

CLOCKF          ; to state 4
CLOCKF          ; to state 2

TRACE_OFF
```

COMBWC02.PDS

```

;*****
;*
;* COMBWC02.PDS - Palasm2 input file for the
;* comment byte write circuit for the utucam project.
;*
;* This PAL is a simply data translator that accepts at
;* input, the four logger signals and the level information:
;*
;*         Logger Signals
;*
;*         LC - Level Captured
;*         WEX - Wait state exited
;*         WEN - Wait state entered
;*         TO - Timer overflowed
;*
;*         Level Information
;*
;*         CPL2_0 - Current Priority Level
;*         LPL2_0 - Last Priority Level
;*
;* It outputs eight signals, D7-D0, which are then inputs
;* to FIFO 3, the comment FIFO. The outputs D7-D0 are
;* related to the inputs by the following relationship.
;*
;*         Inputs           Outputs
;*
;*         LC WEX WEN TO   D7-D0 (hex)
;*         -----
;*
;*         x   x   x   1   00h
;*         x   x   1   0   00h
;*         x   1   0   0   00h
;*         1   0   0   0   CPL,LPL info
;*         0   0   0   0   FFh
;*
;* The CPL and LPL info, when LC is active with no other input
;* is in the following format
;*
;*         Comment Byte
;*
;*         7       6       5       4       3       2       1       0
;*         -----
;*         - 0 - CPL2 - CPL1 - CPL0 - 0 - LPL2 - LPL1 - LPL0 -
;*         -----
;*
;* version 2 of this circuit simply changes the four inputs
;* LC WEX WEN and TO from active high inputs to active
;* low inputs.
;*
;*         Inputs           Outputs

```

```

;*          /LC /WEX /WEN /TO   D7-D0 (hex)
;*          -----
;*
;*          x   x   x   1       00h
;*          x   x   1   0       00h
;*          x   1   0   0       00h
;*          1   0   0   0       CPL,LPL info
;*          0   0   0   0       FFh
;*
;*****

```

```

TITLE          COM_BYTE_WRT
PATTERN        WRITE CIRCUIT
REVISION       02
AUTHOR         KEN BAKER
COMPANY        UNIVERSITY OF TENNESSEE
DATE           SEPT. 11,1990

```

```

CHIP          COM_BYTE_WRT          PAL16L8

```

```

;PINOUT  1    2    3    4        5    6    7    8    9    10

          /LC  /WEX  /WEN  /TIMOV CPL2 CPL1 CPL0 LPL2 LPL1 GND

;PINOUT  11   12   13   14   15  16  17  18  19  20

          LPL0 D7  D6  D5  D4  D3  D2  D1  D0  VCC

```

```

STRING      NOIN      '/TIMOV*/WEN*/WEX*/LC'
STRING      LCONLY    '/TIMOV*/WEN*/WEX*/LC'

```

EQUATIONS

```

D7          = NOIN
D7.TRST     = VCC

```

```

D6          = NOIN
            + LCONLY*CPL2
D6.TRST     = VCC

```

```

D5          = NOIN
            + LCONLY*CPL1
D5.TRST     = VCC

```

```

D4          = NOIN
            + LCONLY*CPL0
D4.TRST     = VCC

```

```

D3          = NOIN
D3.TRST     = VCC

```



```

D2      = NOIN
        + LCONLY*LPL2
D2.TRST = VCC

D1      = NOIN
        + LCONLY*LPL1
D1.TRST = VCC

D0      = NOIN
        + LCONLY*LPL0
D0.TRST = VCC

;*****
;*   END OF THE EQUATIONS SECTION
;*****

SIMULATION

;*****
;*   Since the Block is all combinational, with only four
;*   inputs, the simulation will run through all possible
;*   combination of inputs and observe the outputs.
;*****

TRACE_ON TIMOV WEN WEX LC CPL2 CPL1 CPL0 LPL2 LPL1 LPL0 D7 D6 D5 D4 D3 D2 D1 D0

SETF /CPL2 /CPL1 /CPL0 /LPL2 /LPL1 /LPL0      ;init CPL,LPL info

SETF /TIMOV /WEN /WEX /LC                      ;sets output to FF

;*****
;*   This next tests to see if the outputs follow CPL and
;*   LPL when they change, and the control inputs are set
;*   to indicate that LC is the only event active in the
;*   minicrosecond.
;*****

SETF /TIMOV /WEN /WEX LC                      ;sets output to CPL,LPL info

SETF /CPL2 /CPL1 CPL0 /LPL2 /LPL1 LPL0      ;Tests CPL,LPL = 1,1
SETF /CPL2 CPL1 /CPL0 /LPL2 LPL1 /LPL0      ;Tests CPL,LPL = 2,2
SETF /CPL2 CPL1 CPL0 /LPL2 LPL1 LPL0        ;Tests CPL,LPL = 3,3
SETF CPL2 /CPL1 /CPL0 LPL2 /LPL1 /LPL0      ;Tests CPL,LPL = 4,4
SETF CPL2 /CPL1 CPL0 LPL2 /LPL1 LPL0        ;Tests CPL,LPL = 5,5
SETF CPL2 CPL1 /CPL0 LPL2 LPL1 /LPL0        ;Tests CPL,LPL = 6,6
SETF CPL2 CPL1 CPL0 LPL2 LPL1 LPL0          ;Tests CPL,LPL = 7,7

SETF CPL2 /CPL1 CPL0 /LPL2 LPL1 LPL0

```

```
;*****
;*   Tests the remaining possible configurations for the
;*   control inputs.
;*****
```

```
SETF /TIMOV /WEN WEX /LC
SETF /TIMOV /WEN WEX LC
SETF /TIMOV WEN /WEX /LC
SETF /TIMOV WEN /WEX LC
SETF /TIMOV WEN WEX /LC
SETF /TIMOV WEN WEX LC
SETF TIMOV /WEN /WEX /LC
SETF TIMOV /WEN /WEX LC
SETF TIMOV /WEN WEX /LC
SETF TIMOV /WEN WEX LC
SETF TIMOV WEN /WEX /LC
SETF TIMOV WEN /WEX LC
SETF TIMOV WEN WEX /LC
SETF TIMOV WEN WEX LC
```

```
TRACE_OFF
```

```
;*****
;*   End of the Simulation Section
;*****
```

CMDBWC02.PDS

```

;*****
;*
;* CMDBWC02.PDS - Palasm2 input file for the
;* command byte write circuit for the utucam project.
;*
;* This PAL is a simply data translator that accepts at
;* input, the four logger signals:
;*
;*          LC - Level Captured
;*          WEX - Wait state exited
;*          WEN - Wait state entered
;*          TO  - Timer overflowed
;*
;* It outputs eight signals, D7-D0, which are then inputs
;* to FIFO 0, the command FIFO. The outputs D7-D0 are
;* related to the inputs by the following relationship.
;*
;*          Inputs          Outputs
;*
;*          LC WEX WEN TO   D7-D0 (hex)
;*          -----
;*
;*          x  x  x  1      00h
;*          x  x  1  0      04h
;*          x  1  0  0      06h
;*          1  0  0  0      02h
;*          0  0  0  0      FFh
;*
;* version 2 of this pal simply has the input signals
;* as active low signals (ie. lc wex wen and to become
;* /LC /WEX /WEN /TO. The signal relationship becomes
;*
;*          INPUTS          OUTPUTS
;*
;*          /LC /WEX /WEN /TO   D7-D0 (HEX)
;*          -----
;*
;*          x  x  x  1      00h
;*          x  x  1  0      04h
;*          x  1  0  0      06h
;*          1  0  0  0      02h
;*          0  0  0  0      FFh
;*
;*****

TITLE          CMD_BYTE_WRT
PATTERN        WRITE CIRCUIT
REVISION       01
AUTHOR         KEN BAKER
COMPANY        UNIVERSITY OF TENNESSEE
DATE           JULY 9,1990

CHIP           CMD_BYTE_WRT          PAL16L8

;PINOUT      1    2    3    4    5    6    7    8    9    10

```

```

        /LC /WEX /WEN /TIMOV NC NC NC NC NC GND
;PINOUT 11 12 13 14 15 16 17 18 19 20
        NC D7 D6 D5 D4 D3 D2 D1 D0 VCC

```

EQUATIONS

```

D7      = /LC*/WEX*/WEN*/TIMOV
D7.TRST = VCC

```

```

D6      = /LC*/WEX*/WEN*/TIMOV
D6.TRST = VCC

```

```

D5      = /LC*/WEX*/WEN*/TIMOV
D5.TRST = VCC

```

```

D4      = /LC*/WEX*/WEN*/TIMOV
D4.TRST = VCC

```

```

D3      = /LC*/WEX*/WEN*/TIMOV
D3.TRST = VCC

```

```

D2      = /TIMOV*WEN
          + /TIMOV*/WEN*WEX
          + /LC*/WEX*/WEN*/TIMOV
D2.TRST = VCC

```

```

D1      = /TIMOV*/WEN*WEX
          + /TIMOV*/WEN*/WEX*LC
          + /LC*/WEX*/WEN*/TIMOV
D1.TRST = VCC

```

```

D0      = /LC*/WEX*/WEN*/TIMOV
D0.TRST = VCC

```

```

;*****
;*   END OF THE EQUATIONS SECTION
;*****

```

SIMULATION

```

;*****
;*   Since the Block is all combinational, with only four

```

```

;*   inputs, the simulation will run through all possible
;*   combination of inputs and observe the outputs.
;*****

```

```
TRACE_ON TIMOV WEN WEX LC D7 D6 D5 D4 D3 D2 D1 D0
```

```

SETF /TIMOV /WEN /WEX /LC
SETF /TIMOV /WEN /WEX LC
SETF /TIMOV /WEN WEX /LC
SETF /TIMOV /WEN WEX LC
SETF /TIMOV WEN /WEX /LC
SETF /TIMOV WEN /WEX LC
SETF /TIMOV WEN WEX /LC
SETF /TIMOV WEN WEX LC
SETF TIMOV /WEN /WEX /LC
SETF TIMOV /WEN /WEX LC
SETF TIMOV /WEN WEX /LC
SETF TIMOV /WEN WEX LC
SETF TIMOV WEN /WEX /LC
SETF TIMOV WEN /WEX LC
SETF TIMOV WEN WEX /LC
SETF TIMOV WEN WEX LC

```

```
TRACE_OFF
```

```

;*****
;*   End of the Simulation Section
;*****

```

DMACNT06.PDS

```

;*****
;*
;*          DMA CONTROLLER FOR THE UTUCAM PROJECT.
;*
;*  THIS CONTROLLER CONTROLS READS FROM THE FIFO MEMORY AND
;*  WRITES TO THE SCSI CHIP, THE 53C80.  IT INTERFACES TO THE
;*  FOLLOWING LINES.
;*
;*  INPUTS
;*
;*  DRQ          - DMA REQUEST LINE ( FROM THE 53C80)
;*  START_DMA    - START SIGNAL      ( FROM THE UPROCCESOR )
;*  DMA_DONE     - END OF DMA SIGNAL ( FROM BYTE COUNTER)
;*
;*  OUTPUTS
;*
;*  /DACK        - DMA HANDSHAKE SIGNAL ( TO THE 53C80)
;*  /IOW         - WRITE SIGNAL        ( TO THE 53C80)
;*  /EOP         - INDICATES END TO DMA TRANSFER ( TO 53C80)
;*  COUNT        - INCREMENTS THE BYTE COUNTER ( TO BYTE CNTR)
;*  RESET_COUNT  - RESETS THE BYTE COUNT ( TO BYTE CNTR)
;*
;*****
;*
;*  THE CONTROLLER LOGIC IS OUTLINED ON THE DAYS OF 5/15/90 AND
;*  AFTER IN MY DESIGN NOTEBOOK.  IT IS INTENDED FOR
;*  IMPLEMENTATION *
;*  IN A PAL16R8.                      (KFB)
;*
;*****
;*  THIS FILE, DMACNT02 DIFFERS FROM DMACNT01 BECAUSE I
;*  INPLEMENTING THE FILE USING BOOLEAN EQUATIONS, NOT
;*  STATE MACHINE DESCRIPTIONS.
;*****
;*  DMACNT03.PDS DIFFERS FROM DMACNT02.PDS BY THE FOLLOWING
;*  FACTORS:
;*
;*      COUNT IS ACTIVE HIGH NOW INSTEAD OF ACTIVE LOW
;*      RESET COUNT IS NOW ACTIVE LOW INSTEAD OF ACTIVE HIGH
;*      /OE WAS ADDED TO THE LIST OF INPUTS, THIS WAS MISSIND
;*      OF THE PREVIOUS PALS, BUT THE SIMULATION
;*      MISSED THE ERROR.
;*****
;*
;*  DMACNT04.PDS DIFFERS FROM 03 BY THE FOLLOWING FACTORS:
;*
;*      I USED THE STATE DESCRIPTION GIVEN IN
;*      PALASM INSTEAD OF THE EQUATIONS.
;*      I ADDED AN INITIAL powerup outputs TO CLEAR THE
;*      DMA COUNTER on powerup.
;*      I added the ability to reset the DMA counter
;*      whenever the the DMA cntrlr is reset
;*
;*****
;*
;*  DMACNT05.PDS DIFFERS FROM 04 BY:
;*

```



```

;*      CORRECTION OF THE RACE CONDITION BETWEEN THE
;*      COUNTER AND THE 74LS138 ON THE DMA BOARD.  IT
;*      DOES SO BY ADDING ANOTHER WAIT STATE BETWEEN
;*      INCREMENTING THE DMA COUNTER AND READING THE
;*      FIFO.
;*
;*****
;*
;*      DMACNT06.PDS DIFFERS FROM 05 BY:
;*
;*      MAPPING OUTPUTS TO STATE VARIABLES AND
;*      USING THE REGISTERS TO SYNCHRONIZE THE
;*      INPUTS (1) DRQ, AND (2) START_DMA
;*
;*      DMACNT06 ALSO USES MIXED MODE OPERATION.  BOTH
;*      STATE AND EQUATION TYPE INPUTS ARE USED.
;*
;*      I ALSO ADDED A FIX TO ALLOW CORRECT RESETING
;*      OF THE DMA BYTE COUNTER.
;*
;*****

TITLE          DMA CONTROLLER PAL
PATTERN        DMA STATE MACHINE CONTROLLER
REVISION       06
AUTHOR         KEN BAKER
COMPANY        UNIVERSITY OF TENNESSEE
DATE           JAN. 30,1991

CHIP           DMAPAL          PAL16R8

;----- PIN SPECIFICATIONS -----

PIN    1      CLOCK           ;INPUT
PIN    2      DRQ             ;INPUT
PIN    3      START_DMA       ;INPUT
PIN    4      DMA_DONE        ;INPUT
PIN    5      /RESET          ;INPUT
PIN    10     GND              ;INPUT
PIN    11     /OE             ;INPUT
PIN    12     /QO             ;OUTPUT
PIN    13     DRQ_SYNC        ;OUTPUT  **** CHANGED IN V06 ****
PIN    14     START_DMA_SYNC  ;OUTPUT  **** CHANGED IN V06 ****
PIN    15     /RESET_COUNT    ;OUTPUT
PIN    16     COUNT           ;OUTPUT
PIN    17     /EOP            ;OUTPUT
PIN    18     /IOW            ;OUTPUT
PIN    19     /DACK           ;OUTPUT
PIN    20     VCC             ;INPUT

```

```
;----- Start of the Equation Section -----
```

```
EQUATIONS
```

```
DRQ_SYNC := DRQ           ;DRQ_SYNC IS REGISTERED DRQ
START_DMA_SYNC := START_DMA ;START_DMA_SYNC IS REG. START_DMA
```

```
;----- Start of the State section -----
```

```
STATE
```

```
MEALY_MACHINE
```

```
START_UP := POWER_UP -> WAIT
```

```
START_UP.OUTF := POWER_UP -> /DACK*/IOW*/EOP*/COUNT*RESET_COUNT
```

```
;----- State Assignment section -----
```

```
WAIT = /DACK*/COUNT*/Q0           ;STATE 1 IN DESIGN
WAIT2 = /DACK*/COUNT*/Q0          ;STATE 23 IN DESIGN
WAIT3 = /DACK*/COUNT*/Q0          ;STATE 23A IN DESIGN
SET_WRITE = DACK*COUNT*/Q0        ;STATE 4 IN DESIGN
DELAY_WRITE = DACK*COUNT*/Q0      ;STATE 4A IN DESIGN
DO_WRITE = DACK*/COUNT*/Q0        ;STATE 5 IN DESIGN
DONEQ = DACK*/COUNT*/Q0           ;STATE 6 IN DESIGN
```

```
;----- NEW TRANSITION EQUATIONS -----
```

```
WAIT      := CON4      -> WAIT2
          + CON2      -> WAIT2
          + CON6      -> WAIT2
          +->          WAIT

WAIT2     := CON6      -> WAIT3
          + RST       -> WAIT
          +->          WAIT2

WAIT3     := RST       -> WAIT
          +->          SET_WRITE

SET_WRITE := RST       -> WAIT
          +->          DELAY_WRITE

DELAY_WRITE := RST     -> WAIT
          +->          DO_WRITE

DO_WRITE  := RST       -> WAIT
          +->          DONEQ

DONEQ     := DMA_OVER  -> WAIT ;if count through then exit
          + CON0      -> WAIT ;if start_dma lost then exit
          + CON4      -> WAIT ;if start_dma lost then exit
          +->          WAIT2 ;continue waiting for DRQ
```

```
;----- STATE OUTPUT EQUATIONS -----
```

```

WAIT.OUTF      := CON1      -> /IOW*/EOP*RESET_COUNT
               + CON3      -> /IOW*/EOP*RESET_COUNT
               + CON5      -> /IOW*/EOP*RESET_COUNT
               + CON7      -> /IOW*/EOP*RESET_COUNT
               + CON2      -> /IOW*/EOP*/RESET_COUNT
               + CON4      -> /IOW*/EOP*/RESET_COUNT
               + CON6      -> /IOW*/EOP*/RESET_COUNT
               + CON0      -> /IOW*/EOP*/RESET_COUNT
               +-->        /IOW*/EOP*RESET_COUNT

WAIT2.OUTF     := CON6      -> /IOW*/EOP*/RESET_COUNT
               + RST       -> /IOW*/EOP*RESET_COUNT
               +-->        /IOW*/EOP*/RESET_COUNT

WAIT3.OUTF     := RST       -> /IOW*/EOP*RESET_COUNT
               +-->        IOW*/EOP*/RESET_COUNT

SET_WRITE.OUTF := RST       -> /IOW*/EOP*RESET_COUNT
               +-->        IOW*/EOP*/RESET_COUNT

DELAY_WRITE.OUTF := CON1    -> IOW*EOP*/RESET_COUNT
               + CON3      -> IOW*EOP*/RESET_COUNT
               + CON5      -> IOW*EOP*/RESET_COUNT
               + CON7      -> IOW*EOP*/RESET_COUNT
               + CON0      -> IOW*/EOP*/RESET_COUNT
               + CON2      -> IOW*/EOP*/RESET_COUNT
               + CON4      -> IOW*/EOP*/RESET_COUNT
               + CON6      -> IOW*/EOP*/RESET_COUNT
               +-->        /IOW*/EOP*RESET_COUNT

DO_WRITE.OUTF  := RST       -> /IOW*/EOP*RESET_COUNT
               +-->        /IOW*/EOP*/RESET_COUNT

DONEQ.OUTF     := RST       -> /IOW*/EOP*RESET_COUNT
               + DMA_OVER  -> /IOW*/EOP*RESET_COUNT
               + CON0      -> /IOW*/EOP*RESET_COUNT
               + CON4      -> /IOW*/EOP*RESET_COUNT
               +-->        /IOW*/EOP*/RESET_COUNT

```

```

;----- Conditions segment -----
CONDITIONS

```

```

CON0 = /DRQ_SYNC*/START_DMA_SYNC*/DMA_DONE*/RESET
CON1 = /DRQ_SYNC*/START_DMA_SYNC*/DMA_DONE*/RESET
CON2 = /DRQ_SYNC*/START_DMA_SYNC*/DMA_DONE*/RESET
CON3 = /DRQ_SYNC*/START_DMA_SYNC*/DMA_DONE*/RESET
CON4 = DRQ_SYNC*/START_DMA_SYNC*/DMA_DONE*/RESET
CON5 = DRQ_SYNC*/START_DMA_SYNC*/DMA_DONE*/RESET
CON6 = DRQ_SYNC*/START_DMA_SYNC*/DMA_DONE*/RESET
CON7 = DRQ_SYNC*/START_DMA_SYNC*/DMA_DONE*/RESET
RST  = RESET
DMA_OVER = DMA_DONE*/RESET

```

```
;----- Beginning of the Simulation -----
```

```
SIMULATION
```

```
TRACE_ON CLOCK DRQ START_DMA DMA_DONE /IOW /EOP /RESET_COUNT /RESET DRQ_SYNC
START_DMA_SYNC /DACK COUNT Q0
SETF OE /DRQ RESET /START_DMA /DMA_DONE      ;SIM OF INITIAL STATE WITH
```

```
                                ;reset active
CLOCKF CLOCK                    ;clock through the first state
CLOCKF CLOCK
SETF /RESET                     ;deactivate the reset input
CLOCKF CLOCK                    ;clock 5 times with no input
CLOCKF CLOCK                    ;proves no wierd transitions
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK
```

```
;*****
;*      pretend that the normal cycle begins
;*      with drq occurring first.  the next block will
;*      test if start_dma begins first
;*****
```

```
SETF /DMA_DONE                  ;RESET DMA_DONE AS If dma was done
CLOCKF CLOCK                    ;show what happens when dma_done is set
SETF DRQ                        ;act as if a normal input cycle
CLOCKF CLOCK                    ;TAKE TO STATE 2
CLOCKF CLOCK                    ;waits in state 2 until drq and start_dma are ok
CLOCKF CLOCK
CLOCKF CLOCK
SETF START_DMA                  ;both start_dma and drq are true to goto st 3
CLOCKF CLOCK                    ;TAKE TO STATE 3
SETF /DRQ                       ;act as if 5380 has reset the drq line
CLOCKF CLOCK                    ;TAKE TO STATE 4
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK                    ;take to state 5
CLOCKF CLOCK                    ;take to state 6
CLOCKF CLOCK                    ;dma_done not active so go to state 2
SETF DRQ                        ;ask for another byte send
CLOCKF CLOCK
CLOCKF CLOCK                    ;goto state 3
SETF DMA_DONE                   ;pretend that the clock output has given back
                                ;a dma_done signal
CLOCKF CLOCK                    ;goto state 4
SETF /DRQ                       ;act as if 5380 reset the drq line
CLOCKF CLOCK                    ;goto state 5
CLOCKF CLOCK                    ;goto state 6
CLOCKF CLOCK                    ;goto state 1 since dma_done is active
CLOCKF CLOCK                    ;reset the counter in state 1
SETF /DMA_DONE                   ;pretend that the dma byte counter is reset
CLOCKF CLOCK                    ;wait for input again
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK
```

CLOCKF CLOCK

```
;*****
;*      this code tests a normal cycle with
;*      start_dma beginning first
;*****
SETF RESET /START_DMA      ;set reset to clear the controller
CLOCKF CLOCK
CLOCKF CLOCK
SETF /RESET START_DMA      ;act as if a normal input cycle
CLOCKF CLOCK               ;TAKE TO STATE 2
CLOCKF CLOCK               ;waits in st 2 til drq and start_dma are both active
CLOCKF CLOCK
CLOCKF CLOCK
SETF DRQ                   ;both start_dma and drq are true to goto st 3
CLOCKF CLOCK               ;TAKE TO STATE 3
SETF /DRQ                  ;act as if 5380 has reset the drq line
CLOCKF CLOCK               ;TAKE TO STATE 4
CLOCKF CLOCK               ;take to state 5
CLOCKF CLOCK               ;take to state 6
CLOCKF CLOCK               ;dma_done not active so go to state 2
SETF DRQ                   ;ask for another byte send
CLOCKF CLOCK               ;goto state 3
SETF DMA_DONE              ;pretend that the clock output has given back
                           ;a dma_done signal
CLOCKF CLOCK               ;goto state 4
SETF /DRQ                  ;act as if 5380 reset the drq line
CLOCKF CLOCK               ;goto state 5
CLOCKF CLOCK               ;goto state 6
CLOCKF CLOCK               ;goto state 1 since dma_done is active
CLOCKF CLOCK               ;reset the counter in state 1
SETF /DMA_DONE              ;pretend that the dma byte counter is reset
CLOCKF CLOCK               ;wait for input again
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK
```

```
;*****
;*      this next code test the reset line
;*      in the middle of a transaction
;*****
SETF START_DMA             ;act as if a normal input cycle
CLOCKF CLOCK               ;TAKE TO STATE 2
CLOCKF CLOCK               ;waits in st 2 til drq and start_dma are both active
CLOCKF CLOCK
CLOCKF CLOCK
SETF DRQ                   ;both start_dma and drq are true to goto st 3
CLOCKF CLOCK               ;TAKE TO STATE 3
SETF /DRQ                  ;act as if 5380 has reset the drq line
CLOCKF CLOCK               ;TAKE TO STATE 4
CLOCKF CLOCK               ;take to state 5
SETF RESET                 ;activate the reset line and take back to state 1
CLOCKF CLOCK               ;goto state 1 since reset is active
SETF /RESET
CLOCKF CLOCK               ;goto state 2 since start_dma is still active
```

```

CLOCKF CLOCK          ;stay in state2
SETF DRQ              ;ask for another byte send
CLOCKF CLOCK          ;goto state 3 because drq and start_dma active
SETF DMA_DONE /DRQ    ;pretend that the clock output has given back
                      ;a dma_done signal
CLOCKF CLOCK          ;goto state 4
SETF /DRQ             ;act as if 5380 reset the drq line
SETF RESET /DMA_DONE
CLOCKF CLOCK          ;goto state 1 for reset
CLOCKF CLOCK
CLOCKF CLOCK          ;goto state 1 since dma_done is active
CLOCKF CLOCK          ;reset the counter in state 1
CLOCKF CLOCK          ;wait for input again
CLOCKF CLOCK

;*****
;*      this code tests a normal cycle with
;*      start_dma beginning first, then dropping
;*      out in the middle of the cycle.
;*****
SETF RESET /START_DMA ;set reset to clear the controller
CLOCKF CLOCK
SETF /RESET START_DMA ;act as if a normal input cycle
CLOCKF CLOCK          ;TAKE TO STATE 2
CLOCKF CLOCK          ;waits in st 2 til drq and start_dma are both active
SETF DRQ              ;both start_dma and drq are true to goto st 3
CLOCKF CLOCK          ;TAKE TO STATE 3
SETF /DRQ             ;act as if 5380 has reset the drq line
CLOCKF CLOCK          ;TAKE TO STATE 4
CLOCKF CLOCK          ;take to state 5
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK
CLOCKF CLOCK
SETF DRQ              ;ask for another byte send
CLOCKF CLOCK          ;goto state 3
SETF /START_DMA       ;pretend that the START_DMA signal has failed
CLOCKF CLOCK          ;goto state 4
SETF /DRQ             ;act as if 5380 reset the drq line
CLOCKF CLOCK          ;goto state 5
CLOCKF CLOCK          ;goto state 6
CLOCKF CLOCK          ;goto state 1 since START_DMA has failed
CLOCKF CLOCK          ;reset the counter in state 1
CLOCKF CLOCK
CLOCKF CLOCK

TRACE_OFF

```

SCSI_CH2.PDS

```

;*****
;*
;*
;*          SCSI INTERFACE CHIP
;*
;*          BY KEN BAKER
;*          MARCH 24,1990
;*
;*  THIS CHIP IS USED TO INTERFACE BETWEEN THE
;*  EXPANSION CONNECTOR ON THE 68HC11 FORTH BOARD AND
;*  THE 5380 FAMILY OF SCSI INTERFACE CHIPS.
;*  It is also used to interface the the 74LS245 that
;*  controls the data bus direction and driving for
;*  interface to the expansion bus of the forth board.
;*
;*
;*
;*****
;*
;*          LIST OF VARIABLES
;*
;*          INPUTS
;*          A15-A10
;*          E
;*          R/W
;*
;*          OUTPUTS
;*          SCSI_CS - SELECTS THE L53C80 SCSI BUS CHIP
;*          IOR - SIGNAL THAT INDICATED READING
;*          IOW - SIGNAL THAT INDICATED WRITING
;*          XBUS_EN - SIGNAL THAT ALLOWS THE DRIVING OF
;*                   THE DATA BUS LINES TO THE HC11.
;*
;*
;*          ALL OUTPUTS ARE ACTIVE LOW, AND ARE INTENDED
;*          TO BE USED AS ACTIVE LOW CHIP SELECTS.
;*          THEY NOT ONLY SELECT CHIPS, BUT WILL ALSO BE
;*          USED FOR AREA SELECTS (IF FOR EXAMPLE, ADDR_B_RD
;*          IS ADDED LATER, IT WILL USE THE SAME LINE ON
;*          THIS PAL BUT WILL USE THE READ WRITE LINE FOR
;*          SELECTION.
;*
;*****
;*****
;*          BEGINNING OF THE TITLE BLOCK
;*****

TITLE      SCSI and XBUS INTERFACE CHIP FOR UTUCAM
PATTERN    SCSI_XBUS_INTERFACE
REVISION   001
AUTHOR     KEN BAKER
COMPANY    UNIVERSITY OF TENNESSEE
DATE       MARCH 24, 1990

```



```

;*****
;*          CHIP SPECIFICATION BLOCK          *
;*****

CHIP          SCSI_XBUS_INT    PAL16L8

;PINS      1      2      3      4      5      6      7      8      9      10
          A15  A14  A13  A12  A11  A10  E    RW    NC    GND

;PINS      11 12 13 14 15
          NC NC NC NC NC

;PINS      16      17      18      19      20
          /XBUS_EN /IOW      /IOR      /SCSI_CS  VCC

;*****
;*          EQUATION DECLARATION SECTION      *
;*****

EQUATIONS

SCSI_CS    =    /A15*A14*/A13*A12*/A11*/A10*E    ;ADDR IS 5000-53FF

SCSI_CS.TRST = VCC                                ;TRISTATE ALWAYS ON

IOR        =    /A15*A14*/A13*A12*/A11*/A10*E*RW ;SCSI INTERFACE
                                ;READ SIGNAL

IOR.TRST = VCC                                ;TRISTATE ALWAYS ON

IOW        =    /A15*A14*/A13*A12*/A11*/A10*E*/RW ;SCSI INTERFACE
                                ;WRITE SIGNAL

IOW.TRST = VCC                                ;TRISTATE ALWAYS ON

XBUS_EN =      E*/A15*A14*/A13*/A12              ;XBUS DATA BUS ENABLE
+              E*/A15*A14*/A13*A12              ;SIGNAL. INTERFACES
+              E*/A15*A14*A13*/A12              ;WITH /G ON LS245.
+              E*/A15*A14*A13*A12
+              E*A15*/A14*/A13*/A12

XBUS_EN.TRST = VCC                                ;TRISTATE ALWAYS ON

;*****
;*          SIMULATION SECTION                *
;*****

SIMULATION

TRACE_ON A15 A14 A13 A12 A11 A10 E RW /SCSI_CS /IOR /IOW /XBUS_EN

SETF /E

;*****
;*          THIS SECTION TESTS IOR,IOW WITH E LOW
;*****

```

```
SETF RW
SETF /RW
```

```
;*****
;*   THIS SECTION TESTS THE CHIP SELECT SIGNALS
;*****
```

```
SETF /A15 /A14 /A13 /A12 /A11 /A10
SETF /A15 /A14 /A13 /A12 /A11 A10
SETF /A15 /A14 /A13 /A12 A11 /A10
SETF /A15 /A14 /A13 /A12 A11 A10
SETF /A15 /A14 /A13 A12 /A11 /A10
SETF /A15 /A14 /A13 A12 /A11 A10
SETF /A15 /A14 /A13 A12 A11 /A10
SETF /A15 /A14 /A13 A12 A11 A10
SETF /A15 /A14 A13 /A12 /A11 /A10
SETF /A15 /A14 A13 /A12 /A11 A10
SETF /A15 /A14 A13 /A12 A11 /A10
SETF /A15 /A14 A13 /A12 A11 A10
SETF /A15 /A14 A13 A12 /A11 /A10
SETF /A15 /A14 A13 A12 /A11 A10
SETF /A15 /A14 A13 A12 A11 /A10
SETF /A15 /A14 A13 A12 A11 A10
SETF /A15 A14 /A13 /A12 /A11 /A10
SETF /A15 A14 /A13 /A12 /A11 A10
SETF /A15 A14 /A13 /A12 A11 /A10
SETF /A15 A14 /A13 /A12 A11 A10
SETF /A15 A14 /A13 A12 /A11 /A10
SETF /A15 A14 /A13 A12 /A11 A10
SETF /A15 A14 /A13 A12 A11 /A10
SETF /A15 A14 /A13 A12 A11 A10
SETF /A15 A14 A13 /A12 /A11 /A10
SETF /A15 A14 A13 /A12 /A11 A10
SETF /A15 A14 A13 /A12 A11 /A10
SETF /A15 A14 A13 /A12 A11 A10
SETF /A15 A14 A13 A12 /A11 /A10
SETF /A15 A14 A13 A12 /A11 A10
SETF /A15 A14 A13 A12 A11 /A10
SETF /A15 A14 A13 A12 A11 A10
SETF A15 /A14 /A13 /A12 /A11 /A10
SETF A15 /A14 /A13 /A12 /A11 A10
SETF A15 /A14 /A13 A12 /A11 /A10
SETF A15 /A14 /A13 A12 /A11 A10
SETF A15 /A14 /A13 A12 A11 /A10
SETF A15 /A14 /A13 A12 A11 A10
SETF A15 A14 /A13 /A12 /A11 /A10
SETF A15 A14 /A13 /A12 /A11 A10
SETF A15 A14 /A13 /A12 A11 /A10
SETF A15 A14 /A13 /A12 A11 A10
SETF A15 A14 /A13 A12 /A11 /A10
SETF A15 A14 /A13 A12 /A11 A10
SETF A15 A14 /A13 A12 A11 /A10
SETF A15 A14 /A13 A12 A11 A10
SETF A15 A14 A13 /A12 /A11 /A10
SETF A15 A14 A13 /A12 /A11 A10
SETF A15 A14 A13 /A12 A11 /A10
SETF A15 A14 A13 /A12 A11 A10
SETF A15 A14 A13 A12 /A11 /A10
SETF A15 A14 A13 A12 /A11 A10
SETF A15 A14 A13 A12 A11 /A10
SETF A15 A14 A13 A12 A11 A10
```

```

SETF A15 A14 /A13 /A12 /A11 /A10
SETF A15 A14 /A13 /A12 /A11 A10
SETF A15 A14 /A13 /A12 A11 /A10
SETF A15 A14 /A13 /A12 A11 A10
SETF A15 A14 /A13 A12 /A11 /A10
SETF A15 A14 /A13 A12 /A11 A10
SETF A15 A14 /A13 A12 A11 /A10
SETF A15 A14 /A13 A12 A11 A10
SETF A15 A14 A13 /A12 /A11 /A10
SETF A15 A14 A13 /A12 /A11 A10
SETF A15 A14 A13 /A12 A11 /A10
SETF A15 A14 A13 /A12 A11 A10
SETF A15 A14 A13 A12 /A11 /A10
SETF A15 A14 A13 A12 /A11 A10
SETF A15 A14 A13 A12 A11 /A10
SETF A15 A14 A13 A12 A11 A10

```

```

; THIS IS THE TEST FOR THE E SIGNAL ON THE SCSI_CS SIGNAL
; ALSO CHECKS THE XBUS_EN SIGNAL

```

```
SETF E
```

```

;*****
;* THIS SECTION TESTS IOR,IOW WITH E HIGH
;*****

```

```

SETF RW
SETF /RW

```

```

;*****
;* THIS SECTION TESTS THE CHIP SELECT SIGNALS
;*****

```

```

SETF /A15 /A14 /A13 /A12 /A11 /A10
SETF /A15 /A14 /A13 /A12 /A11 A10
SETF /A15 /A14 /A13 /A12 A11 /A10
SETF /A15 /A14 /A13 /A12 A11 A10
SETF /A15 /A14 /A13 A12 /A11 /A10
SETF /A15 /A14 /A13 A12 /A11 A10
SETF /A15 /A14 /A13 A12 A11 /A10
SETF /A15 /A14 /A13 A12 A11 A10
SETF /A15 /A14 A13 /A12 /A11 /A10
SETF /A15 /A14 A13 /A12 /A11 A10
SETF /A15 /A14 A13 /A12 A11 /A10
SETF /A15 /A14 A13 /A12 A11 A10
SETF /A15 /A14 A13 A12 /A11 /A10
SETF /A15 /A14 A13 A12 /A11 A10
SETF /A15 /A14 A13 A12 A11 /A10
SETF /A15 /A14 A13 A12 A11 A10
SETF /A15 A14 /A13 /A12 /A11 /A10
SETF /A15 A14 /A13 /A12 /A11 A10
SETF /A15 A14 /A13 /A12 A11 /A10
SETF /A15 A14 /A13 /A12 A11 A10
SETF /A15 A14 /A13 A12 /A11 /A10
SETF /A15 A14 /A13 A12 /A11 A10
SETF /A15 A14 /A13 A12 A11 /A10
SETF /A15 A14 /A13 A12 A11 A10

```

```

SETF /A15 A14 /A13 A12 A11 /A10
SETF /A15 A14 /A13 A12 A11 A10
SETF /A15 A14 A13 /A12 /A11 /A10
SETF /A15 A14 A13 /A12 /A11 A10
SETF /A15 A14 A13 /A12 A11 /A10
SETF /A15 A14 A13 /A12 A11 A10
SETF /A15 A14 A13 A12 /A11 /A10
SETF /A15 A14 A13 A12 /A11 A10
SETF /A15 A14 A13 A12 A11 /A10
SETF /A15 A14 A13 A12 A11 A10
SETF A15 /A14 /A13 /A12 /A11 /A10
SETF A15 /A14 /A13 /A12 /A11 A10
SETF A15 /A14 /A13 /A12 A11 /A10
SETF A15 /A14 /A13 /A12 A11 A10
SETF A15 /A14 /A13 A12 /A11 /A10
SETF A15 /A14 /A13 A12 /A11 A10
SETF A15 /A14 /A13 A12 A11 /A10
SETF A15 /A14 /A13 A12 A11 A10
SETF A15 /A14 A13 /A12 /A11 /A10
SETF A15 /A14 A13 /A12 /A11 A10
SETF A15 /A14 A13 /A12 A11 /A10
SETF A15 /A14 A13 /A12 A11 A10
SETF A15 /A14 A13 A12 /A11 /A10
SETF A15 /A14 A13 A12 /A11 A10
SETF A15 /A14 A13 A12 A11 /A10
SETF A15 /A14 A13 A12 A11 A10
SETF A15 A14 /A13 /A12 /A11 /A10
SETF A15 A14 /A13 /A12 /A11 A10
SETF A15 A14 /A13 /A12 A11 /A10
SETF A15 A14 /A13 /A12 A11 A10
SETF A15 A14 /A13 A12 /A11 /A10
SETF A15 A14 /A13 A12 /A11 A10
SETF A15 A14 /A13 A12 A11 /A10
SETF A15 A14 /A13 A12 A11 A10
SETF A15 A14 A13 /A12 /A11 /A10
SETF A15 A14 A13 /A12 /A11 A10
SETF A15 A14 A13 /A12 A11 /A10
SETF A15 A14 A13 /A12 A11 A10
SETF A15 A14 A13 A12 /A11 /A10
SETF A15 A14 A13 A12 /A11 A10
SETF A15 A14 A13 A12 A11 /A10
SETF A15 A14 A13 A12 A11 A10

```

```

;*****
;* CHECKS THE IOR LINE *
;*****

```

```

SETF /A15 A14 /A13 A12 /A11 /A10 E RW

```

```
;*****
;*  THIS SECTION MAKES A REALISTIC LOOK      *
;*  AT THE TIMING FOR A SCSI WRITE AND READ  *
;*****
```

```
;*  INITIALIZATION
```

```
SETF /E RW A15 A14 A13 A12 A11 A10
SETF /E
SETF /E
SETF /E
SETF /E
SETF /E
SETF /E
SETF /E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
```

```
;*  AN ARBITRARY CONDITION
```

```
SETF /E
SETF /E
SETF /E /A15 /A14 /A13 /A12 /A11 /A10 RW
SETF /E
SETF /E
SETF /E
SETF /E
SETF /E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
```

```
;*  SETS THE REAL CONDITION FOR SCSI ACCESS READ
```

```
SETF /E
SETF /E
SETF /E /A15 A14 /A13 A12 /A11 /A10 RW
SETF /E
SETF /E
SETF /E
SETF /E
SETF /E
SETF E
SETF E
SETF E
SETF E
```

```
SETF E
SETF E
SETF E
SETF E
```

```
;*   SETS THE REAL CONDITION FOR SCSI ACCESS WRITE
SETF /E
SETF /E
SETF /E /A15 A14 /A13 A12 /A11 /A10 /RW
SETF /E
SETF /E
SETF /E
SETF /E
SETF /E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
```

```
;*   RETURNS TO SOME OTHER CONDITION
SETF /E
SETF /E
SETF /E /A15 /A14 /A13 /A12 /A11 /A10 RW
SETF /E
SETF /E
SETF /E
SETF /E
SETF /E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
SETF E
```

```
;*****
;*   END OF THE SIMULATION      *
;*****
TRACE_OFF
```

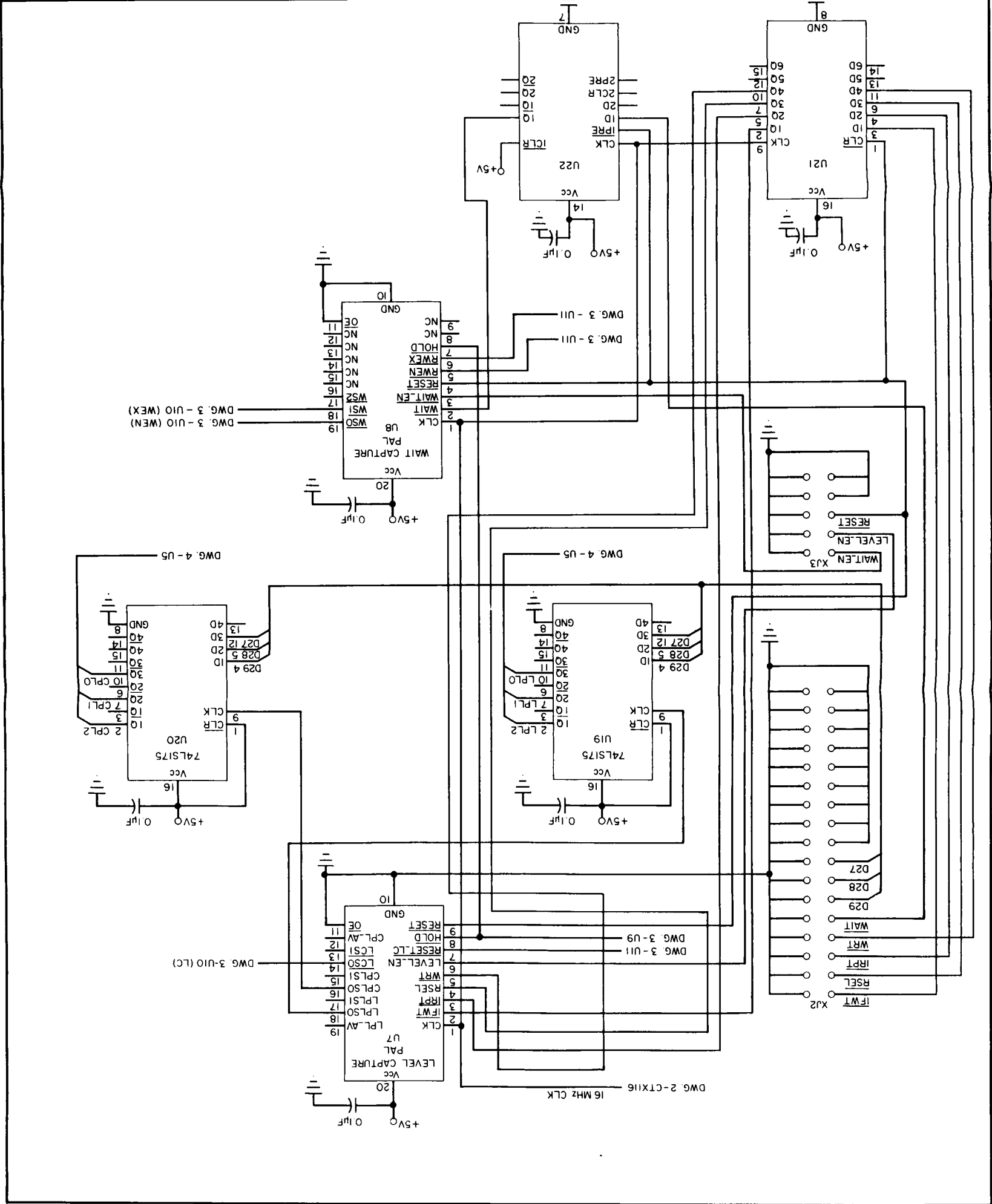
APPENDIX E

MONITOR HARDWARE SCHEMATICS

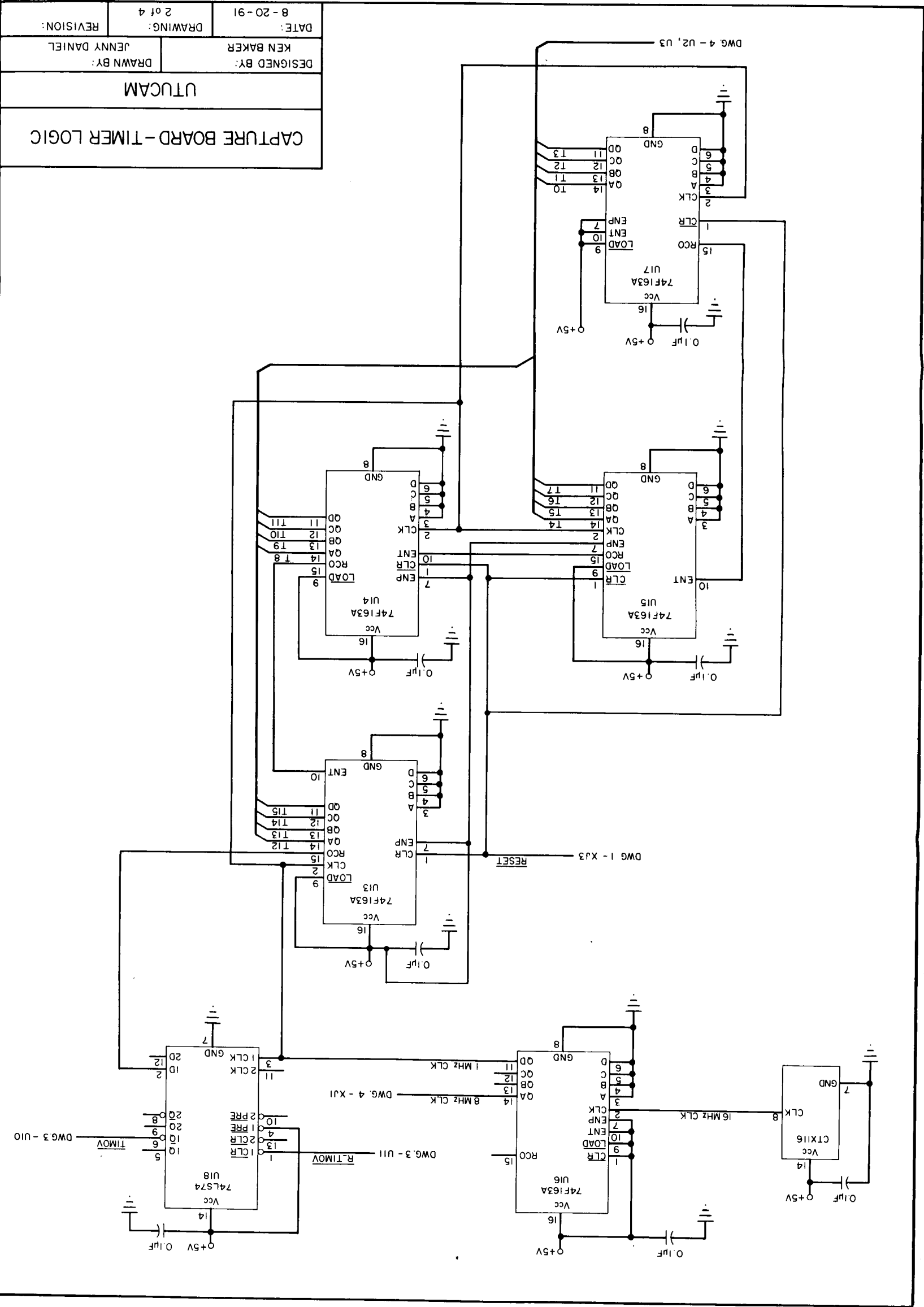
This appendix contains the drawings for the hardware boards implemented for the Priority Event Monitor. Three boards were built for the present implementation of the Priority Event Monitor and they are expressed in the following six drawings, listed below.

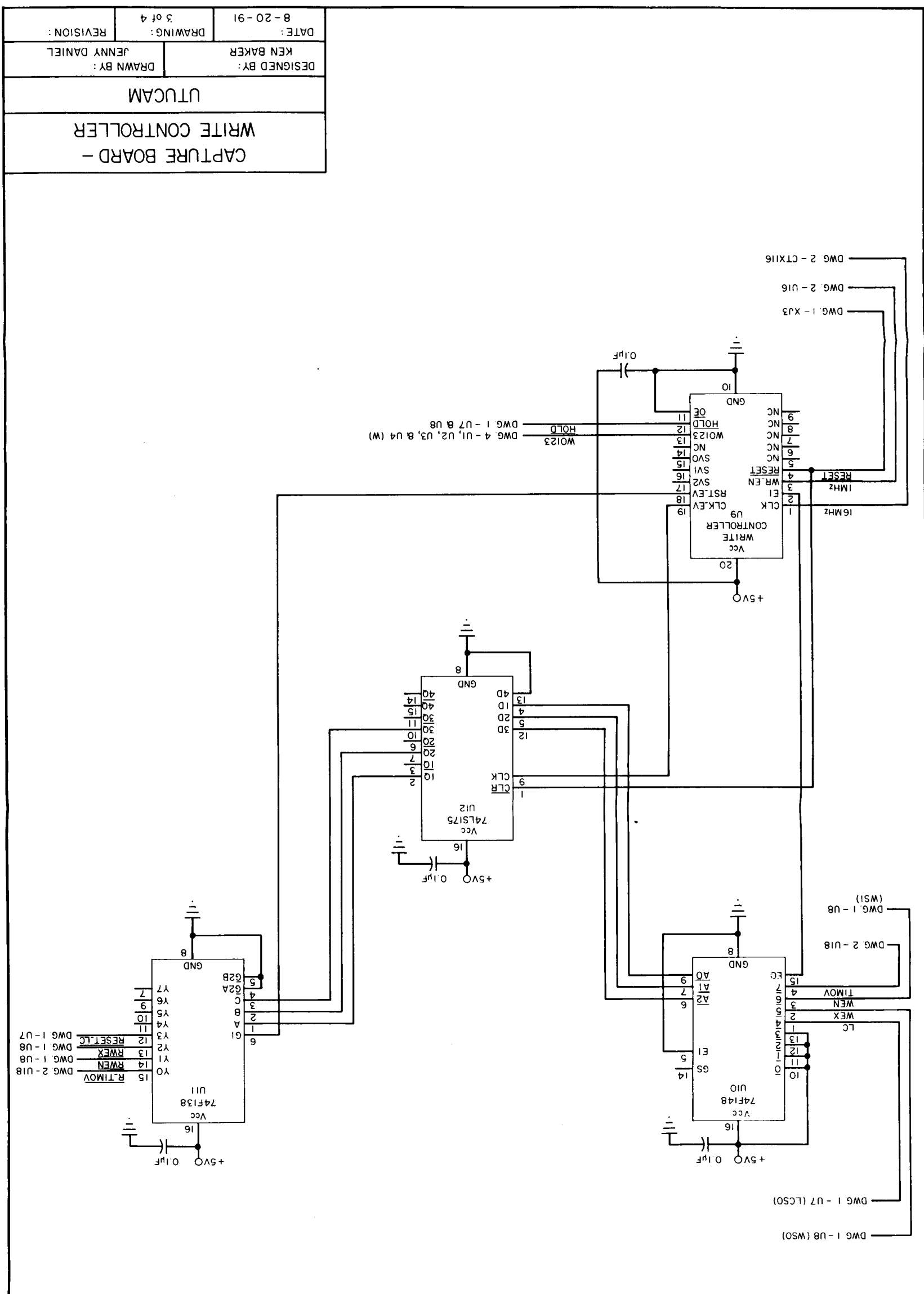
Capture Logic	Covers Priority Level and Wait Capture
Timer Logic	Covers Clocking and Timer Overflows
Write Controller	Covers Write Controller
FIFO Logic	Covers FIFO and Command and Comment Converters
DMA Controller	Covers the DMA Controller
SCSI Board	Covers the SCSI Bus Interface

CAPTURE BOARD -		UTUCAM	
DESIGNED BY KEN BAKER		DRAWN BY JENNY DANIEL	
DATE 9-16-91		DRAWING 1 of 4	
		REVISION 1	

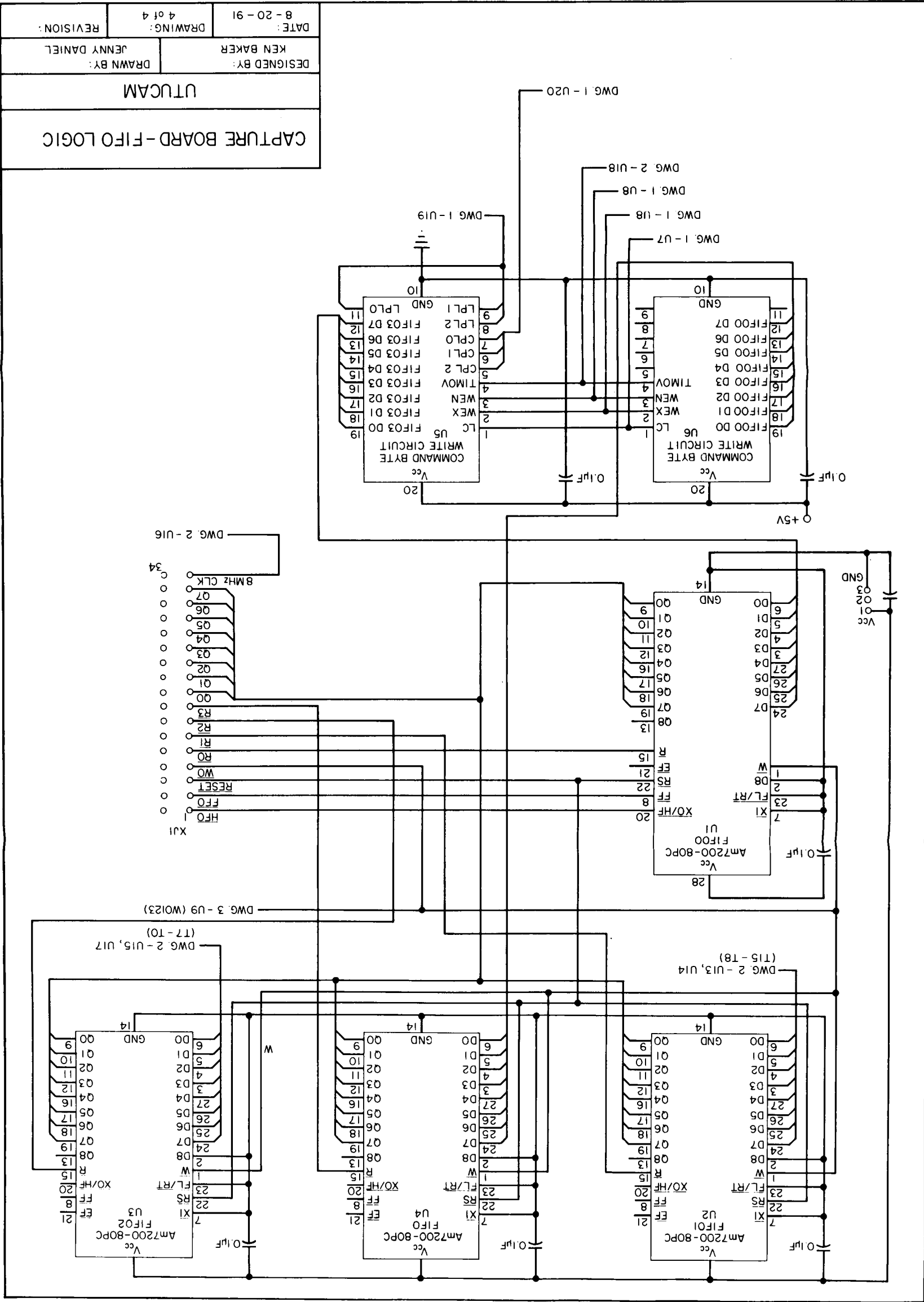


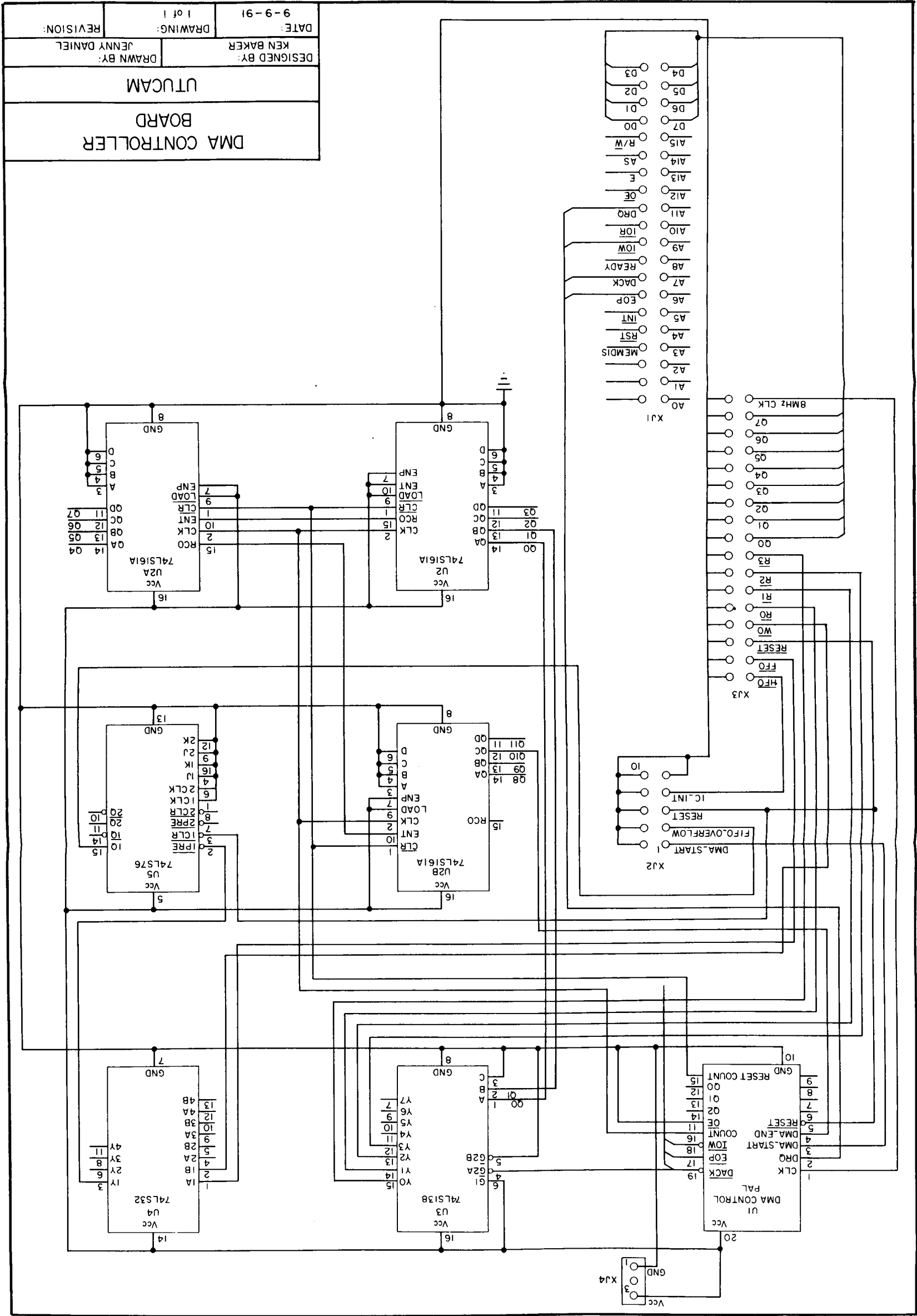
CAPTURE BOARD - TIMER LOGIC	
UTUCAM	
DESIGNED BY: KEN BAKER	DRAWN BY: JENNY DANIEL
DATE: 8 - 20 - 91	DRAWING: 2 of 4
REVISION:	

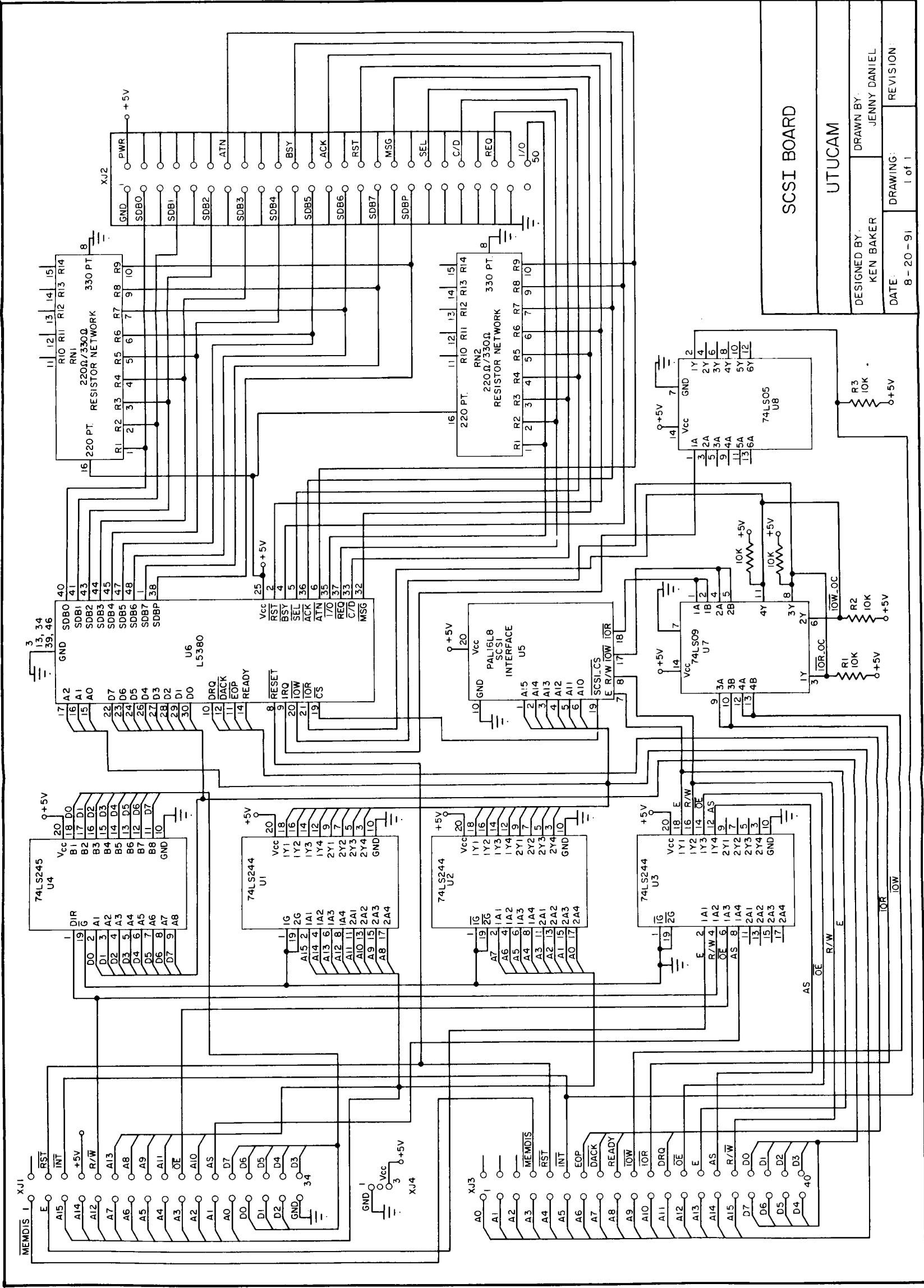




CAPTURE BOARD - FIFO LOGIC		UTUCAM		DESIGNED BY: KEN BAKER	DRAWN BY: JENNY DANIEL	DATE: 8-20-91	DRAWING: 4 of 4
				REVISION:			







VITA

Kenneth Franklin Baker, Jr. was born in Columbia, Tennessee on September 29, 1966 to Kenneth and Glenda Baker. After graduating from North Side High School, Jackson, Tennessee in 1984, he attended the University of Tennessee, enrolled in the Electrical and Computer Engineering Department. His core courses consisted of digital and ASIC design courses, and in June of 1988, he received his Bachelor of Science degree in Electrical Engineering.

In August of 1988, He entered graduate school at the University of Tennessee, seeking a Master of Science degree in Electrical Engineering. During his studies, he served the Electrical and Computer Engineering Department as a graduate assistant, teaching laboratory courses in small computer systems and digital and ASIC design. Also during his graduate career, he served as Chairman of the student branch of the Institute of Electrical and Electronic Engineers. His core courses of study were advanced digital design and VLSI design, with some material on probability and statistics, digital signal processing, and linear algebra.

He is a member of the Institute of Electrical and Electronic Engineers, Eta Kappa Nu, and Tau Beta Pi.