THE DESIGN AND IMPLEMENTATION OF AN EMBEDDED SYSTEMS LABORATORY EMPHASIZING COMPLEX LOGIC DEVICES

BY

JONATHAN EDWARD GREENLAW

B.S., University of Illinois, 1990

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering in the Graduate College of the University of Illinois at Urbana–Champaign, 1996

Urbana, Illinois
WE HEREBY RECOMMEND THAT THE THESIS BY

JONATHAN EDWARD GREENLAW

ENTITLED THE DESIGN AND IMPLEMENTATION OF AN EMBEDDED SYSTEMS
LABORATORY EMPHASIZING COMPLEX LOGIC DEVICES

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

Committee on Final Examination†

Chairperson

Director of Thesis Research

Head of Department

† Required for doctor's degree but not for master's.
ACKNOWLEDGEMENTS

The list of people who helped make this project possible includes far too many people to list here. However, I wish to specifically thank several very important people who made this project possible. I would like to thank Professor Timothy Trick and Professor Narayana Rao for having the faith and vision to support ECE 311. In addition, I would like to thank Intel for their immense financial and technical support.

I would like to thank Professor Pen-Chung Yew and Professor William Sanders who gave me the freedom to pursue the many areas of interest that are now part of ECE 311. I would also like to thank Dr. Ricardo Uribe for being my mentor and friend as well as my advisor.

I would like to thank Anthony Carrico and Kevin Nickels. Their help and guidance were instrumental in the success of ECE 311. The effort they expended goes well above and beyond the call of duty. I would also like to thank Ramesh Yarlagadda and Wendy Bishop who have been my partners in solving many of the ECE 311 problems.

Finally, and most importantly, I would like to thank my wife, Peggy, and the rest of my family for their support for my less than orthodox method of graduate study.
# TABLE OF CONTENTS

## CHAPTER | PAGE

1 INTRODUCTION ...................................................................................................... 1  
1.1 Embedded System Design in an Educational Environment ............................ 1  
1.2 Overview of the ECE 311 Embedded System ..................................................... 1  
1.3 Overview of Experiments .................................................................................... 1  
1.4 The Xilinx Keypad Scanner ................................................................................ 2  

2 EMBEDDED PROGRAMMING ............................................................................... 4  
2.1 Introduction ......................................................................................................... 4  
2.2 Polled I/O ............................................................................................................ 4  
2.3 Interrupt I/O ......................................................................................................... 5  
  2.3.1 Configuring the 80960 ................................................................................ 6  
  2.3.2 Configuring the 8259A ................................................................................ 6  
  2.3.3 Interrupt handlers ........................................................................................ 7  

3 EV80960SX SCSI ADAPTER CARD ...................................................................... 10  
3.1 Introduction ....................................................................................................... 10  
3.2 EV80960SX SCSI Adapter Card Functional Overview ....................................... 10  
3.3 EV80960SX SCSI Adapter Card Datapath ........................................................... 11  
3.4 SCSI Interface Signal Definitions ..................................................................... 13  
  3.4.1 Address and data signals ........................................................................... 13  
  3.4.2 XBUS datapath control signals .................................................................. 14  
  3.4.3 DMA datapath control signals .................................................................. 15  
  3.4.4 DMA buffer control signals ...................................................................... 16  
  3.4.5 AM33C93A control signals ...................................................................... 16  
3.5 State Machine Implementation .......................................................................... 17  
  3.5.1 Internal state machine signals ................................................................... 17  
  3.5.2 Arbitration and decoding .......................................................................... 18  
  3.5.3 Core state machine and controller ............................................................ 19  
3.6 Transactions ....................................................................................................... 19  
  3.6.1 The 80960--DMA buffer transaction ....................................................... 19  
  3.6.2 The 80960--AM33C93A SCSI chip transaction ....................................... 22  
  3.6.3 The DMA setup transaction ..................................................................... 23  
  3.6.4 The AM33C93A--DMA buffer transaction ............................................ 23  
  3.6.5 The AM33C93A SCSI chip--SCSI bus transaction .................................. 24  

4 EV80960SX LCD ADAPTER CARD ..................................................................... 26  
4.1 Introduction ....................................................................................................... 26  

fv
4.2 EV80960SX LCD Adapter Card Functional Overview ................................... .27
4.3 EV80960SX LCD Adapter Card Datapath ........................................................ 27
4.4 EV80960SX LCD Adapter Signals ................................................................... 27
4.5 EV80960SX LCD Controller Signals ............................................................... 28
  4.5.1 LCD timing signals. ................................................................................... 29
  4.5.2 Video buffer signals ............................................................................... 29
4.6 EV80960SX LCD Controller Modules ............................................................. 29
  4.6.1 Clock generator module ................................................................. 31
  4.6.2 Priority encoder module ........................................................................... 31
  4.6.3 Output buffer module ............................................................................... 32
  4.6.4 Video buffer refresh module ..................................................................... 32
  4.6.5 Row transfer module ................................................................................ 32
  4.6.6 XBUS transfer module ............................................................................. 33
4.7 Printed Circuit Board Features .......................................................................... 33

5 ECE 311 MONITOR SOFTWARE .......................................................................... 34
5.1 ECE 311 Monitor Overview .............................................................................. 34
5.2 ECE 311 Monitor Debugging Routines ............................................................ 34
  5.2.1 DMA buffer class debugging routines ...................................................... 35
  5.2.2 Video buffer class debugging routines ..................................................... 35
  5.2.3 Interrupt class debugging routines ............................................................ 36
  5.2.4 SCSI class debugging routines ................................................................. 36
5.3 ECE 311 Monitor Kernel Routines ...................................................................... 37
  5.3.1 EV80960SX SCSI adapter device driver .................................................. 37
  5.3.2 Audio playback routines ........................................................................... 38
  5.3.3 File system structure .............................................................................. 38

6 CONCLUSIONS ................................................................................................. .40
6.1 Summary of the Laboratory Experiments ........................................................ .40
6.2 Achievements ..................................................................................................... 40
6.3 Future Directions ................................................................................................ 41

APPENDIX A EV80960SX SCSI ADAPTER VHDL SOURCE CODE ...................... 42
  A.1 Arbitrator/Decode PAL VHDL Listing ..................................................... 42
  A.2 Core State Machine PAL VHDL Listing ................................................ 45
  A.3 Controller PAL VHDL Listing ................................................................. 49
  A.4 DMA Counter PAL VHDL Listing ............................................................... 53

APPENDIX B XBUS TIMING DIAGRAM ................................................................. 55
  B.1 XBUS Read Timing Diagram .................................................................... 55

APPENDIX C EV80960SX LCD ADAPTER VHDL SOURCE .............................. 56
  C.1 Clock Generator Module VHDL Listing ................................................. 56
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Defined SCSI Adapter Transactions</td>
<td>11</td>
</tr>
<tr>
<td>3.2 XBUS Memory Map</td>
<td>14</td>
</tr>
<tr>
<td>3.3 MODE Bits Mapping</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Defined LCD Adapter Transactions</td>
<td>27</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 ECE 311 Embedded System</td>
<td>2</td>
</tr>
<tr>
<td>2.1 80960 Configuration Code</td>
<td>7</td>
</tr>
<tr>
<td>2.2 8259A Configuration Code</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Interrupt Service Routine</td>
<td>9</td>
</tr>
<tr>
<td>3.1 SCSI Adapter Datapath</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Arbitrator State Machine</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Core State Machine</td>
<td>21</td>
</tr>
<tr>
<td>3.4 State Transitions for a Group 0 SCSI Command</td>
<td>25</td>
</tr>
<tr>
<td>3.5 State Transitions for a Group 1 SCSI Command</td>
<td>25</td>
</tr>
<tr>
<td>4.1 LCD Adapter Datapath</td>
<td>28</td>
</tr>
<tr>
<td>4.2 LCD Controller Datapath</td>
<td>30</td>
</tr>
<tr>
<td>5.1 C Structure for <code>cmd</code></td>
<td>34</td>
</tr>
<tr>
<td>5.2 Structure of Command Sent to <code>scsi_io()</code></td>
<td>38</td>
</tr>
<tr>
<td>E.1 XBUS Connector Interface</td>
<td>102</td>
</tr>
<tr>
<td>E.2 Arbitrator and Core State Machine</td>
<td>103</td>
</tr>
<tr>
<td>E.3 Controller and DMA Counter</td>
<td>104</td>
</tr>
<tr>
<td>E.4 DMA Buffer</td>
<td>105</td>
</tr>
<tr>
<td>E.5 SCSI Bus Interface</td>
<td>106</td>
</tr>
<tr>
<td>E.6 Local XBUS Buffers</td>
<td>107</td>
</tr>
<tr>
<td>E.7 AM33C93 SCSI Controller</td>
<td>108</td>
</tr>
<tr>
<td>E.8 Decoupling Capacitors</td>
<td>109</td>
</tr>
<tr>
<td>E.9 XBUS Interface and Arbitrator</td>
<td>111</td>
</tr>
<tr>
<td>E.10 Watchdog Timer</td>
<td>112</td>
</tr>
<tr>
<td>E.11 Xilinx Core</td>
<td>113</td>
</tr>
<tr>
<td>E.12 VRAM, Xchecker Interface, and LCD Interface</td>
<td>114</td>
</tr>
<tr>
<td>E.13 Test Points</td>
<td>115</td>
</tr>
<tr>
<td>E.14 EPROM and Decoupling Capacitors</td>
<td>116</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1 Embedded System Design in an Educational Environment

The design of embedded systems may best be described as the art of making subsystems with different communication protocols exchange data with each other. In this area, complex logic devices are extremely useful. When subsystems have different protocols, a translator is necessary to transmit and receive data. These translators may be as simple as a few discrete logic equations or as advanced as a complex state machine. With the ease of state machine design and logic optimization, programmable logic devices (PLDs) and synthesis tools make this task much simpler.

1.2 Overview of the ECE 311 Embedded System

The ECE 311 embedded system uses an EV80960SX board for the core microprocessor and support peripherals. Added to this system are a SCSI card and an LCD card. The SCSI card is based on the AM33C93A SCSI chip from Advanced Micro Devices. The only SCSI peripheral attached to the SCSI adapter is a CD-ROM drive. The LCD card is based on a Sharp display using a Xilinx 4005 field programmable gate array (FPGA) as a controller. The external bus of the EV80960SX board (XBUS) has a non-buffered extension to allow up to three external peripherals to be attached. Figure 1.1 shows a block diagram of the overall system.

1.3 Overview of Experiments

In the Microcomputer Design Laboratory (ECE 311) at the University of Illinois, students are given the task of constructing an autonomous system given the protocols by which certain subsystems exchange data. The subsystems studied include Small Computer Systems Interface (SCSI), Dynamic Random Access Memory (DRAM), Direct Memory Access (DMA),
parallel interface, and interrupt controller. The software topics include polled input/output (I/O) and interrupt driven I/O.

These topics are covered in the four laboratory assignments: A Xilinx keypad scanner, a virtual printer, a SCSI adapter for the EV80960SX board, and a liquid crystal display (LCD) adapter for the EV80960SX board. The software topics are discussed in Chapter 2. The SCSI adapter is covered in detail in Chapter 3. Chapter 4 describes the LCD adapter.

1.4 The Xilinx Keypad Scanner

The first experiment is an introduction to the architecture and design tools for FPGAs. The students build a controller to scan a keypad and to display the value of the key pressed.
Although the topics of hardware description languages, synthesis, digital simulation, and verification are introduced in this experiment, its main purpose is to introduce the tools for the fourth experiment. In addition, the first experiment allows enough lecture time to adequately introduce the second experiment.
CHAPTER 2
EMBEDDED PROGRAMMING

2.1 Introduction

The second experiment in ECE 311 is the design of a virtual printer. Text is sent from the parallel port of a personal computer to the EV80960SX board. The application program then displays the data and allows the user to modify the data in some way. In addition to a virtual printer, the student must design an application using interrupts to count seconds. The student is then asked to build on the knowledge gained from the virtual printer and timer by creating an original application.

Unlike traditional computer systems, embedded systems require the programmer to be familiar with the central processing unit (CPU) and I/O processors in order to develop efficient code. Many I/O processors can be categorized based on the method used to interact with the processor. The three most common methods are polled I/O, interrupt driven I/O, and DMA. Of these three, polled I/O and interrupt I/O require programming beyond simple device configuration.

The only tool used for this experiment is a cross-compiler for the 80960. The tools are used to write C and assembly language code that will be downloaded into the EV80960SX board.

2.2 Polled I/O

Polled I/O is being used when the processor queries the status of a device. When the status indicates that data have to be exchanged, the program's execution is altered to service the device. The main disadvantage of using polled I/O is that the processor wastes cycles when there is no need for data exchange. Also, the service interval must be frequent enough to prevent the device from overflowing. The advantage of polled I/O is that it is relatively easy to program.
In ECE 311, polled I/O is used to control a serial port and a parallel port. The project's goal is to receive data from the parallel port for display on a serial terminal. The outer polling loop tests the value of the parallel port busy signal (PPBUSY). When active, a routine to read the data and acknowledge the transfer is called. For the serial port, a status register is polled to determine if data are ready to be exchanged.

To meet the setup and hold times of the parallel and serial ports, a software delay loop has to be developed. An example of one used for polling consists of two add instructions, a multiply instruction, and a comparison instruction. The loop is completed twenty times for each unit of delay. The number of units is passed as an argument to the delay code. The average delay for one unit is between twenty and thirty microseconds. An exact delay is difficult to construct due to pipeline interlocking and cache hits and misses.

For this experiment, a program is designed to read an entire file into a buffer and perform any of a set of user commands on the data. A personal computer is used to send a file through its printer port to the EV80960SX board. As the data are read, they are placed into a linked list. The advantage of using a linked list is that memory is requested only when needed. The efficient use of memory in an embedded system is extremely important.

2.3 Interrupt I/O

The main disadvantage of polled I/O is that it requires CPU time in order to determine if data are available. Since most of the polls will indicate an idle state, the CPU is wasting most of the time polling. One solution is to use interrupt driven I/O. In this type of I/O, the CPU is free to perform other tasks until a device signals the need to transfer data. The CPU can then suspend the current process, transfer the data, and then resume the suspended process. In this manner, the CPU does not waste cycles performing memory accesses that deliver no data.
The final part of this software experiment is to design a counter that increments once a second. The application program must configure the 80960, a timer, and an interrupt controller. In addition, the application program must install an interrupt service routine (ISR).

2.3.1 Configuring the 80960

Before any interrupts can be serviced, the 80960 must be configured. Figure 2.1 is a list of steps necessary to install a pointer for the ISR into the 80960 interrupt table. The first step is to locate where the interrupt table is located in memory. By sending an interrupt access control (IAC) message to the CPU, a pointer to the interrupt table is placed into the \textit{system_base} structure. The address of the ISR is then copied into the interrupt table. The final step is to configure the 80960 for interrupt request and acknowledge mode.

2.3.2 Configuring the 8259A

To increase the number of interrupt lines available to the 80960 on the EV80960SX board, an Intel 8259A programmable interrupt controller has been added. The 8259A is a general purpose interrupt controller used with the 8086 series of microprocessors from Intel. The code in Figure 2.2 illustrates how to configure the 8259A.

Configuring the 8259A is a two-part process. The first part specifies the configuration through the use of initialization command words (ICWs). For this experiment, the 8259A is configured for edge triggered interrupts and normal end of interrupt mode. Finally, a vector into the 80960 interrupt table is loaded. The second part uses operation command words (OCWs) to set the interrupt mask. The interrupt mask allows the 8259A to filter interrupt requests from other devices.
iac_struct iac;
unsigned int *int_table;
unsigned int *prcb;
unsigned int system_base[];

/* Store System Base IAC */
iac.message_type = 0x80;

/* place address of buffer */
iac.field3 = (unsigned int)system_base;

/* issue the IAC message */
send_iac((int)&iac);

/* move PRCB to prcb pointer */
prcb = (unsigned int *)system_base[1];

/* Interrupt table is here */
int_table = (unsigned int *)prcb[5];

/* set interrupt vectors */
int_table[EVSX_VEC_TIMER_32A+1] = (unsigned int)timer_int_isr;

/* initialize 80960SA Interrupt register for the 8259A */
interrupt_register_write(0xff000808);

Figure 2.1 80960 Configuration Code

2.3.3 Interrupt handlers

One of the more difficult aspects of interrupt driven I/O is the design of ISRs. An ISR must be capable of servicing the device, yet short enough to prevent other processes from being locked. Figure 2.3 is a complete interrupt handler for use with an Intel 82C54 timer/counter.

Since the 80960 has global registers and an interrupt can occur at any time, the first step for any ISR is to save the global registers. If the global registers are not saved, then the program that was interrupted would suddenly have corrupt data. The next task is to send an acknowledge to the 8259A.
int level;
volatile unsigned char *icwl, *icw2, *icw4, *ocwl, *ocw2;

/* ICW4 needed, single, edge triggered */
*icw1 = (unsigned char)(BIT4|SNGLINEED4);

/* 8259 3:LSBs */
*icw2 = (unsigned char)EVSX_VEC_BASE_8259;

/* 8086 mode, normal EOI, non-buffered */
*icw4 = (unsigned char)uP86;

/* Disable all interrupts */
*ocw1 = (unsigned char)0xFF;

/* Acknowledge all interrupts */
for (level = 0; level < 8; ++level)  
  *ocw2 = (unsigned char)(SPEC_EOI | level);

/* Enable just the interrupts wanted */
*ocw1 = (unsigned char)(0x7F);

Figure 2.2 8259A Configuration Code.

The EV80960SX board has some custom circuitry to prevent multiple timer interrupts from overloading the 80960. Therefore, the next step is to reset this circuit to allow the timer to resume operation.

The application program locks a semaphore that should be released once a second. Once the semaphore is released, the application program updates a display and locks the semaphore again. The final step of this ISR is to release the semaphore so that the application program may proceed.
ldconst 64, r4
add sp, r4, sp

/* Save the global registers */
stq g0, -64(sp)
stq g4, -48(sp)
stq g8, -32(sp)
stt g12, -16(sp)

/* Acknowledge interrupt controller */
ldconst 0x67, g1
lda 0x28000000, g2
stb g1, (g2)

/* Clear timer interrupt */
lda 0x24000006, g2
ldob (g2), g3

/* Release timer semaphore */
ldconst 0x00, g1
stb g1, _timer_semaphore

/* Restore the global registers */
ldq -64(sp), g0
ldq -48(sp), g4
ldq -32(sp), g8
ldr -16(sp), g12
ret

Figure 2.3 Interrupt Service Routine
CHAPTER 3
EV80960SX SCSI ADAPTER CARD

3.1 Introduction

The third experiment in ECE 311 is the design of a SCSI adapter for the EV80960SX board. The SCSI adapter will be used to interface to a CDROM drive to read data and play audio. The controller, glue logic, and interconnections are installed on a printed circuit board. The purpose of this experiment is to introduce students to the advantages and disadvantages of using existing large scale integrated (LSI) components to assemble a system.

The steps in digital system design include partition, design, and test. Since the interconnections of the glue logic are already specified, the student must contend with the design and test portions. As part of the design phase, students must read and understand timing diagrams, design glitch-free state machines, and effectively use simulations. Once the initial design work has been completed, the student must then learn how to debug a circuit using a logic analyzer and how to interpret incorrect results obtained from a software monitor. The process of design and test is an iterative one that eventually leads to a stable solution.

The programmable array logic chips (PALs) in the SCSI adapter are programmed using Warp VHDL from Cypress Semiconductor. This package includes simulation and synthesis tools for use with a 22V10 PAL. Since the 22V10 is not a very complex device, students must learn how to write VHDL code that is not only functionally correct, but can be synthesized into the limited space available.

3.2 EV80960SX SCSI Adapter Card Functional Overview

The EV80960SX SCSI adapter card consists of five interfaces. The first interface is between the Intel 80960 microcontroller via the XBUS of the EV80960SX and the 2K by 16 static random access memory (SRAM) buffer. Unless otherwise stated, all interaction between
the 80960 and the SCSI adapter is via the XBUS. Also, the 2K by 16 SRAM buffer will be referred to as the DMA buffer. The XBUS is a buffered extension of the internal bus of the EV80960SX. Since the EV80960SX is designed around the 80960, the XBUS cycle is similar to the bus cycle of the 80960. The second interface is between the AM33C93A and the DMA buffer. The third interface is between the 80960 and the registers of the AM33C93A. The fourth interface is between the 80960 and the control logic of the DMA controller. The fifth interface is between the AM33C93A and the SCSI bus.

A transaction is defined as the transfer of data over any of the major interfaces. Table 3.1 lists all of the defined transactions for the SCSI adapter.

Table 3.1 Defined SCSI Adapter Transactions

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>80960&lt;-DMA Buffer</td>
<td>Transfer data between the 80960 and the DMA buffer</td>
</tr>
<tr>
<td>80960-&gt;SCSI Register</td>
<td>Transfer control data and status information between the 80960 and the AM33C93A</td>
</tr>
<tr>
<td>DMA Setup</td>
<td>Reset DMA counters and setup DMA mode</td>
</tr>
<tr>
<td>AM33C93A&lt;-DMA Buffer</td>
<td>Transfer data between the AM33C93A into the DMA buffer</td>
</tr>
<tr>
<td>AM33C93A-&gt;SCSI Bus</td>
<td>Transfer data between a SCSI device into the AM33C93A</td>
</tr>
</tbody>
</table>

3.3 EV80960SX SCSI Adapter Card Datapath

The SCSI adapter consists of three major datapaths (see Figure 3.1). The first datapath of the SCSI adapter is the XBUS. For the ECE 311 system, the XBUS buffers of the EV80960SX are externally enabled at runtime to pass the control and data signals to external hardware. The XBUS is intended to interface with only one card. However, by controlling the data path externally using local buffers, several devices may be placed on an external bus structure. In order for the 80960 to access the external hardware, the XBUS is memory mapped into the two regions
Figure 3.1 SCSI Adapter Datapath
0x30000000 to 0x307FFFFF and 0x30800000 to 0x30FFFFFF. The EV80960SX decodes the upper four bits of its internal address bus. If the request is in the XBUS memory range, the lower twenty-four address bits and a region bit are passed over the XBUS. Each card then determines whether or not to respond to the current request by decoding the two highest address bits (XA[23:22]) and the region bit [1]. The second datapath is the DMA channel used by the AM33C93A to exchange data with the DMA buffer. The third datapath is the SCSI bus used by the AM33C93A and the CD-ROM drive. The SCSI bus is a single ended, narrow architecture.

3.4 SCSI Interface Signal Definitions

The control and datapath signal lines of the SCSI interface can be grouped into five categories: address and data signals, XBUS datapath control signals, DMA datapath control signals, DMA buffer control signals, and AM33C93A control signals. These signals are routed to four 22V10 PALs, two CY7C128 2K by 8 SRAMs, six 74LS245 octal bus transceivers, the XBUS, the AM33C93A, and a 7416 open-collector buffer. The four 22V10 PALs function as arbitrator, state machine, controller, and DMA counter, respectively. All signals preceded with an asterisk are active low.

3.4.1 Address and data signals

The XA[23:22] address bits are presented to an arbitrator by the EV80960SX. The arbitrator decodes these bits and activates the appropriate interface. Table 3.2 shows how the address bits are mapped to four of the interfaces. The ADDRESS[11:1] bits represent the local bus shared by the XBUS (XA[11:1]) and the DMA counter (DMA_ADDR[11:1]). The XBUS and DMA control signals are required to place the local XBUS buffers or the DMA counter in a high impedance state to prevent local bus conflicts. The DATA[15:0] bits represent the local bus shared by the XBUS data bits (XD[15:0]) and the AM33C93A data bits (SD[7:0]).
Table 3.2 XBUS Memory Map

<table>
<thead>
<tr>
<th>XA23</th>
<th>XA22</th>
<th>Subsystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>80960 ↔ AM33C93A Registers</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>80960 ↔ SRAM</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>DMA Setup</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>LCD (No SCSI card transaction)</td>
</tr>
</tbody>
</table>

3.4.2 XBUS datapath control signals

Every 80960 memory access requiring the XBUS begins by asserting *XCSO and ends by issuing *XREADY. The *XCSO signal is an input to the arbitration logic. When active, *XCSO indicates that an 80960 cycle in the XBUS range of the memory map has been issued.

The *READY signal indicates when the data on the local bus are ready for latching. The *XREADY signal indicates to the EV80960SX that the data on XD[15:0] are ready for latching by the 80960. This signal is a buffered version of *READY [2].

The XBUS datapath is controlled by several different signals. The *XB_ENA signal is an input to the local XBUS buffers on the SCSI adapter. When active, the other XBUS datapath control signals are valid and the local XBUS buffers are enabled. This signal must be inactive for the DMA counter to drive ADDRESS[11:1] and for the AM33C93A to drive DATA[15:0]. Because the AM33C93A is an eight-bit device, the DMA buffer must be byte addressable. However, the 80960 has a sixteen-bit datapath, so the DMA buffer must also be word addressable. The *XBE0 signal enables the lower byte, which implies that DATA[7:0] should be valid with respect to the 80960. The *XBE1 signal enables the upper byte, which implies that DATA[15:8] should be valid with respect to the 80960. The XWRITE signal indicates the direction of data flow for an XBUS cycle [1].

The XRESET signal is a global control signal for the four PALs and the AM33C93A. When active, this signal indicates that the EV80960SX is in a reset state. All PALs and the
AM33C93A are reset to a known start-up state. The XRESET signal is also routed to an open-collector buffer in order to reset the SCSI bus.

3.4.3 DMA datapath control signals

All AM33C93A DMA transactions begin by issuing a *DRQ to the arbitration logic. The AM33C93A activates this signal when it has to exchange data with the DMA buffer. The *DAK signal is an input to the AM33C93A. When a byte of data has been stored into the DMA buffer, the control PAL issues a *DAK to indicate the end of a DMA cycle [3].

As with the XBUS datapath, various signals are required to control the DMA datapath. The DMA_BUSY signal is an input to the DMA counter. When active, the other DMA datapath control signals are valid, the DMA counter can drive ADDRESS[11:1], and the AM33C93A can drive DATA[15:0]. This signal must be inactive for the local XBUS buffers to drive ADDRESS[11:1] and DATA[15:0]. The *DMA_BHE signal is an input to a 74LS245 buffer. When active during a DMA cycle, this signal configures the datapath to copy the eight-bit data from the AM33C93A to the upper byte of the DMA buffer. Specifically, *DMA_BHE configures the datapath such that SD[7:0] is copied to DATA[15:8]. The SD[7:0] data bits are always available to DATA[7:0]. The DMA_WRITE signal is an input to a 74LS245 buffer responsible for copying SD[7:0] to DATA[8:15]. This signal indicates the direction of data flow between the AM33C93A and the DMA buffer.

The DMA_COUNT signal is an input to the DMA counter. Because the PAL used to implement the DMA counter can only drive ADDRESS[11:2], DMA_COUNT must pulse every four DMA cycles to increment the DMA counter PAL correctly. The DMA_RESET signal sets the DMA counter to zero.
3.4.4 DMA buffer control signals

Since the 80960 and the AM33C93A both access the DMA buffer, they must share its control signals. In order to be byte and word accessible, the DMA buffer requires two eight-bit SRAMs. The control signals \( *\text{SRAM\_CS}0 \) and \( *\text{SRAM\_CS}1 \) must be active before the DMA buffer can be written or read. The \( *\text{SRAM\_CS}0 \) signal enables the lower byte of the DMA buffer to use \( \text{DATA}[7:0] \). Likewise, the \( *\text{SRAM\_CS}1 \) signal enables the upper byte of the DMA buffer to use \( \text{DATA}[15:8] \) [4].

The only other two signals necessary to control the DMA buffer are \( *\text{WE} \) and \( *\text{OE} \). The \( *\text{WE} \) signal is an input to the DMA buffer indicating that \( \text{DATA}[15:0] \) should be latched into the address specified by \( \text{ADDRESS}[11:1] \) [4]. This signal is also connected to \( *\text{RE} \) of the AM33C93A to indicate the direction of data flow between the 80960 and AM33C93A [3]. The \( *\text{OE} \) signal is an input to the DMA buffer indicating that the contents of the DMA buffer specified by \( \text{ADDRESS}[11:1] \) should be placed on \( \text{DATA}[15:0] \) [4]. The \( *\text{OE} \) signal is also connected to \( *\text{WE} \) of the AM33C93A to indicate the direction of data flow between the 80960 and the AM33C93A [3].

3.4.5 AM33C93A control signals

The AM33C93A must be enabled before it can be accessed by the 80960 or access the DMA buffer. The \( *\text{SCSI\_CS} \) signal is an input to the AM33C93A that enables the \( \text{SD}[7:0] \) bits. Data are either latched from or presented to \( \text{DATA}[15:0] \) based on the value of \( *\text{OE} \) and \( *\text{RE} \) [3].

The AM33C93A multiplexes address and data on \( \text{SD}[7:0] \). The only other address bit is \( A0 \). When \( A0 \) is low, \( \text{DATA}[7:0] \) specifies the address of an internal register. When \( A0 \) is high, \( \text{DATA}[7:0] \) represents a value to be stored in the previously specified register of the AM33C93A [3].
3.5 State Machine Implementation

Each transaction of the SCSI adapter is mapped into one state machine. When data have to be transferred via an interface, a state machine cycle begins. Each transaction acquires the necessary resources and directs the relevant portion of the datapath. Since the interfaces share resources and are mutually exclusive in operation, the transactions can be mapped onto each other. The state machine is partitioned among four 22V10 PALs. The PALs are partitioned as arbitrator, core state machine, controller, and DMA counter.

3.5.1 Internal state machine signals

Figure 3.1 shows the detailed datapath including the partitioning of the state machine among four 22V10 PALs. Because the state machine has to be partitioned among four PALs, several additional signals are required to coordinate the functions of the arbitrator, core state machine, controller, and DMA counter.

Once the various devices contend for the local bus and a bus master is selected, the START signal from the arbitrator PAL is issued to the core state machine and controller. The core state machine and the controller then latch the MODE[1:0] bits. The MODE[1:0] bits indicate which transaction the master device has requested. Table 3.3 shows the mapping of the MODE[1:0] bits to transactions.

<table>
<thead>
<tr>
<th>MODE[1:0]</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>80960→AM33C93A Registers</td>
</tr>
<tr>
<td>01</td>
<td>80960↔DMA Buffer</td>
</tr>
<tr>
<td>10</td>
<td>DMA Buffer Setup</td>
</tr>
<tr>
<td>11</td>
<td>AM33C93A↔DMA Buffer</td>
</tr>
</tbody>
</table>

After the core state machine completes its cycle, the DONE signal is sent to the arbitrator. This signal indicates to the arbitrator that the current transaction has ended and that contention for the local bus may begin again.
Due to a shortage of outputs on the 22V10 PAL, the core state machine passes its current state to the controller PAL. The controller PAL then activates the appropriate control signals. The SCSI adapter must synchronize XBUS control signals with the internal processor clock of the 80960. The 80960 uses a 16 MHz, two-phase clocking scheme. The CLK16 signal is derived by dividing the 32 MHz system clock by two and synchronized with XRESET [1].

Since the 80960 is word and byte addressable with a 16 bit data bus, there can be no address bit 0. To increment the DMA counter properly, the control PAL uses the DMA_BYTE state variable to pulse DMA_COUNT every four DMA cycles.

### 3.5.2 Arbitration and decoding

Before any transaction involving the SCSI adapter can begin, the arbitration logic must grant mastership to the appropriate device. Arbitration is handled by a 22V10 PAL. A simple state machine alternates priority between the 80960 and AM33C93A operating in DMA mode. The arbitrator begins in an idle state waiting for either subsystem to request the use of the local bus via the control signal *XS0 or *DRQ. In the initial state, the 80960 is favored over the AM33C93A. If both subsystems request the bus at the same time, the 80960 is granted mastership of the local bus. Subsequently, the arbitrator enables the local XBUS buffers via *XB_ENA. If the AM33C93A is the only subsystem to make a request, then it is granted local bus mastership and the DMA Address PAL is enabled via DMA_ENA. After a transaction occurs, the arbitrator enters a second idle state. In this state, the AM33C93A is favored in the resolution of local bus access conflicts. Figure 3.2 is a state diagram for the arbitrator. The only exception is the AM33C93A ↔ SCSI transaction. In this case, the AM33C93A handles all phases of arbitration and selection on the SCSI bus, but the AM33C93A must still go through the arbitration logic to transfer data into the DMA buffer.

The transitions in Figure 3.2 represent conditions that must be met in order to change state. These conditions are actually Boolean equations derived from the signals defined in
Section 3.4. The equations for each condition and the resultant outputs are defined in Appendix A.1. The default transitions occur when an unexpected or reset condition occurs.

3.5.3 Core state machine and controller

Figure 3.3 shows the basic states available to implement each transaction. Each transaction has different timing needs, so not all states are necessary for each transaction class. Unlike the arbitrator state machine, the core state machine does not loop back on itself. Therefore, each transition is equal to 1 clock cycle or 31.2 ns. The setup and hold times for each particular transaction are constructed using these 31.2 ns blocks of time. The output of the core state machine is sent to the controller PAL. The controller PAL decodes the current state of the core state machine and activates the appropriate control lines. Because the state latched into the controller PAL is one clock cycle behind the current state of the core state machine, the controller has to use more elaborate starting and ending conditions to synchronize with the core state machine. The output and transition equations for the core state machine are defined in Appendix A.2. The output and transition equations for the controller are defined in Appendix A.3.

3.6 Transactions

The SCSI adapter has five major classes of transactions. Each transaction is defined by a unique set of state sequences and control signals.

3.6.1 The 80960→DMA buffer transaction

The XBUS is supposed to be an extension of the 80960 processor bus of the EV80960SX Evaluation Board. The main difference is the demultiplexing of the address and data busses. The bus timing cycle is very similar to the 80960 [2] except that the *XCS0 signal does not always transition to the inactive state. This behavior is attributed to the memory mapping method used in the EV80960SX. Because the address lines become precharged, the multiplexer that
Figure 3.2 Arbitrator State Machine
Figure 3.3 Core State Machine
controls *XCS0 remains active until the next address cycle. The important side effect is that consecutive memory references are not reliable. Appendix B.1 shows the modified XBUS timing cycle.

The sequence of states for the 80960→DMA buffer read transaction is
Idle→D_{a1}→D_{b1}→R_{a0}→R_{b0}. For a read cycle, states D_{a1} and D_{b1} provide 62 ns for the DMA buffer to present valid data as required by the manufacturer’s time constraints. State R_{b0} is used to provide 32 ns for the DMA buffer to move to a high impedance state [4].

The sequence of states for the 80960→DMA buffer write transaction is
Idle→D_{a1}→D_{b1}→R_{a0}→R_{b0}. For a write cycle, states D_{a1} and D_{b1} provide 62 ns for the XBUS to meet the setup and hold time constraints required by the DMA buffer. As with the read transaction, state R_{a0} is used to provide 32 ns for the DMA buffer to move to a high impedance state [4].

Either *SRAM_CS0 or *SRAM_CS1 must be active for the D_{b1} and R_{a0} states in order to access either the lower or the upper byte, respectively. The *XB_ENA signal is asserted for all cycles in order to pass data, address, and control signals to and from the EV80960SX. The *READY signal is asserted during the R_{a0} and R_{b0} cycles in order to signal the 80960 that the transaction is ending. During the write transaction, *READY indicates to the 80960 that the current memory cycle should end. For a read transaction, *READY indicates that the 80960 should latch DATA[15:0] on Φ_b [2]. The XWRITE signal is used by the controller to activate *OE for the read transaction or *WE for the write transaction. These two signals must be asserted in conjunction with *SRAM_CS0 or *SRAM_CS1 for proper DMA buffer access.

3.6.2 The 80960→AM33C93A SCSI chip transaction

The AM33C93A uses the same basic timing diagram presented in Figure 3.3. The main difference is that the number of required wait states and data states is significantly higher. Another difference is that unlike the DMA buffer transactions, the read and write transactions
with the AM33C93A are not symmetric. Thus, the sequence of states that the core state machine follows is different for read and write transactions.

The AM33C93A is configured for processor indirect addressing mode. The timing characteristics require *SCSI_CS to be active for 120 ns for a write cycle and 180 ns for a read cycle [3]. The wait state sequence $W_{a0} \rightarrow W_{b0} \rightarrow W_{a1} \rightarrow W_{b1} \rightarrow W_{a2} \rightarrow W_{b2}$ provides 187 ns of delay during which time *SCSI_CS is active. The rest of the sequence $D_{a1} \rightarrow D_{b1} \rightarrow R_{a0} \rightarrow R_{b0}$ provides the data latching cycles and recovery cycles required by the 80960 [2]. The write sequence is $W_{a2} \rightarrow W_{b2} \rightarrow D_{a1} \rightarrow D_{b1}$ to activate *SCSI_CS for 124 ns. The sequence $R_{a0} \rightarrow R_{b0}$ provides the necessary recovery time for the 80960 to release the XBUS. The XWRITE signal is used to differentiate between a read and a write transaction. From the initial idle state, the core state machine will move to state $W_{a0}$ if the transaction is a read and to state $W_{a2}$ if it is a write.

3.6.3 The DMA setup transaction

Before the AM33C93A can transfer data into the DMA buffer, the DMA controller must be initialized. The DMA counter can only be reset when the DMA controller is configured. Therefore, the DMA controller must be configured for read or write operation for each SCSI request transferring data. The transaction to configure the DMA controller is similar to any other XBUS request. The DMA_DIR signal is set equal to XWRITE. However, since the controller PAL can latch XWRITE with zero wait states, the core state machine can move immediately to the data and recovery cycles.

3.6.4 The AM33C93A → DMA buffer transaction

Since the AM33C93A has no pins dedicated to addressing in the DMA mode, the address bits for the DMA buffer have to be generated by an external PAL. Since these address bits also share the local bus with the 80960, the controller must enable either the DMA counter or the local XBUS buffers. The DMA counter is enabled by the DMA_COUNT signal from the con-
controller and increments every cycle of the XBUS clock. The definitions of the output equations for the DMA counter are found in Appendix A.4.

Because the AM33C93A is an eight-bit peripheral, the DMA controller must send the data bits to either the high byte or the low byte of the DMA buffer. This routing is accomplished by alternating the DMA_BHE signal with every cycle of the *DRQ. Once the DMA controller has been configured for a read or write operation and the AM33C93A properly programmed, data can be moved into the DMA buffer.

3.6.5 The AM33C93A SCSI chip—SCSI bus transaction

The AM33C93A is designed to operate SCSI devices with a minimum of overhead. It can be programmed to execute an entire SCSI request and notify the host processor only at the end of the request or if the request fails [3]. In the ECE 311 system, the AM33C93A acts as a SCSI initiator, which originates an operation. The CD-ROM drive acts as a SCSI target, which performs the operation [5].

The SCSI standard defines the following phases: Bus Free phase, indicating that no device is currently using the bus; Arbitration phase, an initiator or target takes control of the SCSI bus; Selection phase, the initiator selects a target to perform some operation; and Information phase, data are transferred via the data bus. An Information phase may be a Command, Data, Status, or Message phase [5].

Each SCSI command is in reality a frame called a Command Data Block (CDB) that is sent to the target. Typical frame lengths are 6 bytes for Group 0 commands, 10 bytes for Group 1 commands, and 12 bytes for Group 5 commands [5]. Figures 3.4 and 3.5 show examples of Group 0 and Group 1 commands.
Figure 3.4 State Transitions for a Group 0 SCSI Command

Figure 3.5 State Transitions for a Group 1 SCSI Command
CHAPTER 4
EV80960SX LCD ADAPTER CARD

4.1 Introduction

The fourth and final experiment in ECE 311 is the design of an LCD adapter for the EV80960SX board. The LCD adapter can be used to load data from a personal computer and display the image on the LCD screen. After basic functionality is achieved, the student is asked to add new features to the adapter. The video random access memory (VRAM), a Xilinx 4005 FPGA, XBUS decoder, and interconnections are installed on a printed circuit board.

Unlike the SCSI adapter, the primary controller is not already designed. The student must now contend with the partitioning problem in digital design. Although the glue logic must be included in the LCD controller design, the added complexity results in more flexibility in design choices. The testing phase also becomes much more complicated. Since most of the internal signals of the FPGA are not readily available, careful design and simulation have a more important role.

The LCD controller is designed using System Architect from Mentor Graphics. The System Architect tool allows the designer to perform graphical hierarchical and finite state machine design. The tool then generates VHDL code for synthesis by a tool called AutologicII. Unlike the WarpVHDL tool, System Architect and AutologicII are not constrained by limited hardware resources. As a result, the student has to contend with bad design choices made by the synthesis tools. In other words, writing VHDL code that simulates correctly is difficult. Writing VHDL code that simulates and synthesizes correctly is more difficult. Writing VHDL code that simulates, synthesizes, and functions correctly is most difficult.
4.2 EV80960SX LCD Adapter Card Functional Overview

The EV80960SX LCD adapter consists of four interfaces. The first interface is between the 80960 and the 512K by 4 VRAM. The 512K by 4 VRAM will subsequently be referred to as the video buffer. The second interface is between the LCD controller implemented on a Xilinx 4005 FPGA and the video buffer. The third interface is between the LCD controller and the LCD. The fourth interface is between the VRAM and the LCD.

The transactions for the LCD adapter are defined in Table 4.1.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>80960&lt;-&gt;VRAM</td>
<td>Transfer data between the 80960 and the VRAM</td>
</tr>
<tr>
<td>LCD Controller&lt;-&gt;VRAM</td>
<td>Generate VRAM refresh signals</td>
</tr>
<tr>
<td>LCD Controller&lt;-&gt;LCD</td>
<td>Generate LCD timing signals</td>
</tr>
<tr>
<td>VRAM&lt;-&gt;LCD</td>
<td>Transfer data between the VRAM and the LCD.</td>
</tr>
</tbody>
</table>

4.3 EV80960SX LCD Adapter Card Datapath

Figure 4.1 shows the three datapaths of the LCD adapter. The first datapath is the XBUS. Refer to Section 3.2 for a discussion on mapping the LCD adapter into the XBUS memory space. The second datapath is between the LCD and the video buffer. The third datapath is between the LCD controller and the LCD.

4.4 EV80960SX LCD Adapter Signals

At the adapter level, the signals can be grouped in one of three categories: XBUS decode, LCD data from the video buffer, and LCD controller. The XBUS operation is similar to the SCSI controller (see Section 3.3). Unlike the SCSI adapter, the decode PAL only has to activate the LCD_ENA signal when an XBUS transaction is requested. The second grouping of signals consists of the data signals SDQ[3:0]. These signals are the four bits shifted from the serial
access memory (SAM) of the video buffer and sent to the LCD. The third grouping of signals includes the LCD controller.

Figure 4.1 LCD Adapter Datapath

4.5 EV80960SX LCD Controller Signals

The LCD controller datapath is shown in Figure 4.2. The control and datapath signals of the LCD adapter can be grouped into two categories. The first category includes timing signals
for the LCD. The second category includes the signals for managing the video buffer.

4.5.1 LCD timing signals

The LCD is 320 by 240 pixels with built-in shift registers, latches, and LCD drivers. Three signals are required to control the displayed image on the LCD: CP2, CPI, S. The CP2 signal is used to latch the four bits on the SDQ[3:0] data lines into a row buffer. The CPI signal indicates that the row buffer has been filled and to transfer 320 bits into the LCD matrix. The S signal is used to signal the LCD that the next CPI corresponds to the first line of the screen [6].

4.5.2 Video buffer signals

In addition to controlling various DRAM and transfer functions, the *RAS provides the latching signal for the row address. The *CAS signal provides the latching signal for the column address as well as enabling the data outputs. The *W signal enables data to be written into the DRAM. To transfer data from the DRAM into the SAM, the *TRG signal must be activated. The DSF signal is used for special functions that are not implemented in the LCD adapter. Since each row of the SAM is 512 bits wide, the SC signal shifts the next four bits onto the SDQ[3:0] data lines. Because the LCD latches 4 bits of data every CP2, SC is connected to CP2.

4.6 EV80960SX LCD Controller Modules

The video buffer is a dynamic, dual ported memory. Data flow from the XBUS into the video buffer DRAM. The first 320 rows of the DRAM are then sequentially shifted into the SAM. Data from the SAM are then shifted into the LCD. The LCD controller is responsible for coordinating these transactions. The LCD controller is a collection of six interacting modules: clock generator, priority encoder, refresh, row transfer, and output buffer. The VHDL source code for each module may be found in Appendix C.
Figure 4.2 LCD Controller Datapath
4.6.1 Clock generator module

The LCD timing signals are necessary not only for the LCD timing transaction but also for synchronizing the other three transactions as well. The clock generator module uses the XBCLK signal to construct a pulse waveform that is low for 532 ns and high for 156 ns. The pulse signal is then switched from the CP2 signal to the CP1 signal after a count of 80 pulses for the duration of one pulse. To compensate for initial conditions, the S signal counts 19440 pulses, then holds S high for one pulse.

4.6.2 Priority encoder module

The priority encoder is responsible for coordinating the following three transactions: refresh, row transfer, and XBUS transfer. Since the DRAM must be refreshed every 8 ms, the refresh operation is the highest priority. The row transfer operation must be synchronized with the CP2 signal in order for a stable image to be displayed. Thus, row transfers are given second highest priority. Since XBUS transfers can be controlled with the *XREADY signal, these transfers are given the lowest priority.

Unlike the SCSI adapter, the LCD adapter uses a rotating priority scheme based on clock cycles rather than on actual device requests. Since there are 512 rows in the DRAM, a row must be refreshed every 15.6 µs [7]. This constraint allows for 20 CP2 cycles/refresh. During the first CP2, the refresh module has control of all of the video buffer control signals. For the next 14 CP2 cycles, the XBUS module has control of the video buffer control signals. The remaining 5 CP2 cycles are used to allow current XBUS transfers to complete, but not to allow any new XBUS transfer to begin. Since CP1 is used to signal a new row in the LCD, the row transfer cycle of the video buffer may execute concurrently with CP1.
4.6.3 Output buffer module

All of the modules have at least two signals in common. To prevent conflict, the output buffer module uses the output of the priority encoder and CP1 to multiplex the three modules onto the appropriate data and control lines.

4.6.4 Video buffer refresh module

Although the refresh module is the most important module, it is the simplest to implement. The video buffer requires *RAS to be held high for 70 ns and to be held low for 100 ns. The *CAS signal must overlap the *RAS transition from high to low [7]. After the priority encoder enables the refresh module, *CAS is set low. Since the default state for *RAS is high, only one pulse is required to meet the 70 ns requirement thus, *RAS is made active. The next pulse makes *CAS inactive, which satisfies the overlap requirement. Four more pulses are then required to meet the 100 ns requirement for *RAS.

4.6.5 Row transfer module

The row transfer module implements a DRAM to serial access memory operation. This module is actually two submodules. The first submodule is a state machine which generates the appropriate control signals. The second submodule switches the row and column addresses such that *RAS and *CAS latch the appropriate addresses.

The sequence for the state machine submodule is to latch the row address using *RAS and to enable the serial register by activating *TRG. The column address is then set to 0x0 by the second submodule and latched by the video buffer using *CAS. The data will not be available until the next CP2. Therefore, this operation is referred to as an early load operation [7]. The use of early load operations increases the difficulty of synchronizing with the S pulse with the other LCD signals.
4.6.6 XBUS transfer module

The XBUS module is responsible for controlling the XBUS data transaction. This module is composed of three submodules. The first module generates phases for proper XBUS operation. The second module is a state machine that generates the proper control signals for the XBUS and the video buffer. The third module switches the column and row addresses. The operation of the XBUS is documented in Chapter 3.

The XBUS module implements a normal read or an early write operation on the video buffer. The sequence for a read or write is basically the same. The row and column addresses are latched as described in Section 4.6.4. The *W signal is active for a write or inactive for a read. The data for a write must be valid prior to *RAS and *CAS transitioning to an inactive state. The data for a read are available for latching shortly after the *RAS and *CAS signals are set high.

4.7 Printed Circuit Board Features

The printed circuit board contains several features to insure reliability and to provide versatility. The first feature is a watchdog timer. If the period of the CPI signal is too long, the drive transistors in the LCD matrix can be permanently damaged. To prevent damage, power for the LCD is routed through a relay. If more than 130 ms pass between CPI pulses, the watchdog circuit opens the relay and power is removed from the LCD. A momentary switch is installed on the board to reset the watchdog circuit.

The LCD controller can be programmed in one of two ways. The first way is through the xchecker interface of the Xilinx chip. The xchecker interface allows design revisions to be loaded into the Xilinx chip from a personal computer. The second way is through an electrically programmable read only memory (EPROM) chip installed on the LCD adapter. A fully functional design can be programmed into the EPROM for embedded applications. A switch is installed to select the method to be used to configure the Xilinx chip.
CHAPTER 5
ECE 311 MONITOR SOFTWARE

5.1 ECE 311 Monitor Overview

The ECE 311 monitor is designed to run on top of MON960, a debugging monitor from Intel. The main advantage of MON960 is that it provides most of the low-level device drivers and system calls for the C programming language. The ECE 311 monitor consists of several debugging routines to facilitate student design. In addition, a primitive file system kernel is implemented to access ISO-9660 format CD-ROMs and to play audio compact discs.

5.2 ECE 311 Monitor Debugging Routines

The ECE 311 monitor has four classes of debugging routines. The SCSI class allows the user to program and to query the AM33C93A. The interrupt class allows the user to configure the interrupts of the EV80960SX board. The DMA buffer class allows the user to exchange data between the EV80960SX board and the DMA buffer on the SCSI adapter. And finally, the video buffer class allows the user to exchange data with the video buffer on the LCD adapter.

Commands are parsed by the routine `get_command(prompt, command)`, which parses user input and places command arguments into the structure `cmd` shown in Figure 5.1. The individual debugging routines are responsible for error and bounds checking.

```c
struct cmd {
    char choice;
    int nargs;
    long arg[3];
};
```

**Figure 5.1** C Structure for `cmd`.
5.2.1 DMA buffer class debugging routines

The DMA buffer is mapped into the address range 0x30400000 to 0x304007FF. The disp_sram(cmd) routine reads one byte from the address specified in cmd and displays the contents on the serial terminal. The mod_sram(cmd) routine is similar to disp_sram(cmd) except that the address specified is set to the value specified in cmd. The fill_sram(cmd) routine is an extension to the display_sram(cmd) command. The address range specified in cmd is filled with a constant value.

In order to stress test the DMA buffer, the brutalize_sram() generates a series of patterns which are written to and read back from the DMA buffer. The value read from the DMA buffer is compared to the expected value written. Any errors are displayed on the serial terminal. The first pattern is to write the values from 0x0 to 0xFF into a single location. The address written is immediately read back. The pattern is repeated in the next address until every location has been exercised. This test is intended to expose timing problems. The next test is to sequentially write one byte into every location. The values are read only after all memory locations are filled. This test is intended to expose any problems related to datapath control. The relevant source code for the DMA buffer routines may be found in Appendix D.1.

5.2.2 Video buffer class debugging routines

The video buffer debugging routines are very similar to the SRAM debugging routines, except that the addressing method is different. The routines are designed such that locations on the LCD screen are specified by row and column. Another difference is that the data for the video buffer are only four bits wide. Thus all data to and from the video buffer have all but the lowest four bits masked. By utilizing the file system primitives built into MON960, data from the hard disk of an external personal computer can be accessed using standard C file descriptors. The routine paint_bitmap_from_hd() can take data in X bitmap format and display it on the LCD screen. The source code for the video buffer routines may be found in Appendix D.2.
5.2.3 Interrupt class debugging routines

The monitor normally runs at the highest possible priority. This means that no other process can interrupt it. The routine `set_priority(int)` lowers the monitor priority level so that interrupts from other devices will be serviced. The `mod_intr_mask(int)` routine enables the 8259A Programmable Interrupt Controller and masks unnecessary interrupts. The XBUS interrupt is the lowest priority interrupt to the EV80960SX board [1]. The relevant source code for the interrupt routine may be found in Appendix D.3.

5.2.4 SCSI class debugging routines

In order to program the AM33C93A, the `mod_scsi_reg(cmd)` routine must make two memory references. The first memory reference writes to address 0x30000000. The AM33C93 loads the value on the data bus, which indicates which register is to be written. The second memory reference writes to address 0x30000002, which loads the value on the data bus into an internal register of the AM33C93A [3].

In order to poll the registers of the AM33C93A, the `dis_scsi_reg(cmd)` routine must again make two memory references. The first memory reference writes to address 0x30000000. The AM33C93A loads the value on the data bus, which indicates which register is to be read. The second memory reference reads from address 0x30000002. The data on the bus are the contents of the register specified in the first memory reference [3].

The purpose of the `mon_dma_init()` routine is to reset the DMA counter and the DMA controller for writing. A memory write to address 0x3000001 is sufficient to program the DMA control logic.

The purpose of the `readsec()` routine is to make extended checks of all of the subsystems on the SCSI adapter. These tests consist of numerous reads of a patterns CD-ROM. The patterns can be determined by an algorithm; thus, the memory in the DMA buffer can be transferred into the EV80960SX board for verification. The test includes constant pattern matching, modulus
pattern matching, and a mixture of constant and modulus pattern matching. The routine begins by initializing the DMA subsystem. The routine then makes four passes reading the first four sectors of the patterns CD-ROM. Each sector is 2 KBytes in length and contains one of the constants 0x00, 0xFF, 0xAA, or 0x55. The routine then makes four passes reading the next four sectors. These sectors can be reconstructed by counting from 0 to 255 modulo n, where n = 0, 64, 128, or 192. The final phase mixes these two data forms for four more passes on the next four sectors. The relevant source code for the SCSI routines may be found in Appendix D.4.

5.3 ECE 311 Monitor Kernel Routines

The ECE 311 monitor has a file system kernel for reading ISO-9660 format CD-ROMs. It also has a set of vendor specific primitives to play audio compact discs.

5.3.1 EV80960SX SCSI adapter device driver

The EV80960SX SCSI adapter card device driver consists of two basic routines. The first is the initialization routine `scsi_init()`. The first task of `scsi_init()` is to configure the 80960 interrupts. This consists of installing the jump addresses into the interrupt handler table, masking all but the SCSI adapter interrupt, and lowering the MON960 interrupt priority to process all interrupts. The second task is to configure the AM33C93A and issue a soft reset to load the new configuration data.

The second routine handles the AM33C93A specific communication. The command structure (see Figure 5.2) passed to `scsi_io()` consists of a generic CDB. The CDB is parsed and loaded into the AM33C93A. The next step is to issue a select and transfer command to the AM33C93A, which then begins the actual process of transferring the command to the CD-ROM drive. The device driver is designed to handle two exception conditions. The first is an unexpected data phase, which indicates that the CDB is returning more bytes than the AM33C93A was programmed to expect. The course of action is to read each new byte until the data flow...
completes. The routine then returns an error. The other condition is an unexpected message phase. This error is usually caused when the CD-ROM drive is not able to return the requested amount of data. In this case that SCSI chip is reprogrammed to expect the message in phase.

The operation completes and an error message is returned. The source code for the file system kernel may be found in Appendix D.5.

```c
struct sctl_io
{
    unsigned target_id; /* IN: SCTL_READ */
    unsigned cdb_length; /* IN */
    unsigned char cdb[16]; /* IN */
    unsigned char size[4]; /* IN */
    unsigned max_msecs; /* IN: milli-seconds before abort */
    unsigned cdb_status; /* OUT: SCSI status */
    unsigned char sense[256]; /* OUT */
    unsigned sense_status; /* OUT: SCSI status */
};
```

**Figure 5.2** Structure of Command Sent to `scsi_io()`

5.3.2 Audio playback routines

The audio routines are dependent on the firmware of the CD-ROM drive. The 80960 can issue various commands to skip tracks and play audio, but the majority of the data is processed locally on the CD-ROM drive. The ECE 311 monitor software constantly polls the CD-ROM to ascertain the time position in the current audio track. This information is displayed on the serial terminal.

5.3.3 File system structure

The file system is built on the ISO-9660 standard. This standard defines information and directory nodes [8]. The software traces the directory nodes until the relevant file is located. The program then follows the data nodes of the file. Each 2K block of the file is read into the EV80960SX board and stored as a one-way linked list. The head pointer to this list is passed
to any other data processing routines for further use. Currently, the ECE 311 monitor can display the text content of a file on the serial terminal and send bitmapped images to the LCD adapter.
CHAPTER 6
CONCLUSIONS

6.1 Summary of the Laboratory Experiments

ECE 311 exposes students to a variety of hardware and software design issues. The experiments build a hierarchy of modern design methods. The first level is the software design. By developing virtual printer software that is dependent on hardware implementation, students are exposed to the issues of hardware/software co-design. The next level consists of using LSI components to build a system. By designing a SCSI adapter, students are exposed to the problems of protocol differences and how to solve these problems using PLDs. At the next level, students are indirectly exposed to the issues relating to application specific integrated circuits (ASICs). By designing the LCD controller, the student directly controls the functionality of the LCD.

6.2 Achievements

The first offering of ECE 311 was in the Fall of 1993. The course was intended to supplement basic logic design by introducing students to the advantages and limitations of PLDs and FPGAs. In this respect, students and their future employers have provided very positive feedback. However, ECE 311 has achieved even higher goals. Instead of providing a series of small unrelated experiments, students are assembling an advanced embedded computer system. In addition, students are learning to use industrial grade tools to solve real world problems.

The ECE 311 computer system provides an efficient vehicle for students to learn real system design issues. Students apply the theoretical aspects of other courses such as logic design and computer architecture in the practical design of a complete system. In addition, students are exposed to difficulties in subsystem design and communication. Specifically, issues regarding timing and specification are addressed that cannot be adequately modeled in simulation. Finally, the utility and flexibility of programmable logic for rapid prototypes are demonstrated.
The experiments are all designed to allow creativity in implementation and functionality. In this regard, the student solutions and independent extensions have far surpassed any expectations. New ideas and challenges to old assumptions are introduced each semester so that the material is intellectually stimulating as well as practical. In terms of educating students, this is the most satisfying goal achieved.

6.3 Future Directions

One of the tasks of embedded system design is to make subsystems with different communication protocols communicate with each other. Although ECE 311 succeeds in this goal, the practical use of the XBUS is very limited. The course should be migrating to more standard interface architectures such as ISA or PCI.

The hardware design for ECE 311 began in 1992. As a result, most of the components are no longer manufactured. Therefore, the LCD adapter and the SCSI adapter should be re-designed to take advantage of modern technology.

Finally, the software tools should continue to keep pace with advancing technology. ECE 311 is currently migrating from schematic-based design to hardware description language design. As new paradigms of digital design are developed, ECE 311 should continue to integrate them into the course.
APPENDIX A
EV80960SX SCSI ADAPTER VHDL SOURCE CODE

The following listings are the source codes for the PALs used in the SCSI adapter. The code was compiled and simulated using the Warp VHDL compiler from Cypress Semiconductor.

A.1 Arbitrator/Decode PAL VHDL Listing

ENTITY arbitrator IS PORT (
  -- Arbitrator inputs/output
  clk, reset, xcs0, done: IN BIT;
  xa24, xa23, drq: IN BIT;
  -- mode0, model, start, xbusy, dma_busy: out bit;
  control_vec: INOUT x01z_vector(4 DOWNTO 0);
  arb_state: INOUT x01z_vector(2 DOWNTO 0);

  -- Definitions for spurious logic condensation
  sram_cs1: IN BIT;
  dma_bhe: OUT BIT;

  ATTRIBUTE pin_numbers OF arbitrator: ENTITY IS
  "xcs0:2 " &
  "done:3 " &
  "xa24:4 " &
  "xa23:5 " &
  "drq:6 " &
  "reset:7 " &
  "sram_cs1:8 " &
  "control_vec(4):23 " &
  "control_vec(3):22 " &
  "control_vec(2):20 " &
  "arb_state(0):19 " &
  "arb_state(1):18 " &
  "control_vec(1):17 " &
  "control_vec(0):16 " &
  "dma_bhe:15 " &
  "arb_state(2):14 ";
END arbitrator;

USE WORK.rtlpkg.ALL;
USE WORK.int_math.ALL;

ARCHITECTURE arch_arb OF arbitrator IS
BEGIN
moore: PROCESS (clk, reset)
  VARIABLE none, xbus: BOOLEAN;
  VARIABLE dma_req, both, release, stay, not_done: BOOLEAN;
  VARIABLE xscsi, xsram, dmamode: BOOLEAN;

BEGIN
IF reset='!' THEN
  arb_state <= "000";
ELSIF (clk'EVENT AND clk='1') THEN
  -- Boolean expression to simplify if statements in the case statements
  none := (xcs0='1' AND drq='1');
  xbus:= (((xcs0='0' AND drq='1') AND done='0') AND 
  (NOT(xa24='1' AND xa23='1')));
  dma_req := ((xcs0='1' AND drq='0') AND done='0');
  both:= (((xcs0='0' AND drq='0') AND done='0') AND 
  (NOT(xa24='1' AND xa23='1')));
  release := (done='1');
  stay := (done='0');
  not_done := (xcs0='0');
  xscsi := (xa24='0' AND xa23='0') AND done='0';
  xsram := (xa24='0' AND xa23='1') AND done='0';
  dmamode := (xa24='1' AND xa23='0') AND done='0';

  -- arb_state:<=
  "000" when (state=idle_x) else
  "001" when (state=idle_d) else
  "010" when (state=cs_x) else
  "011" when (state=cs_d) else
  "100" when (state=wait);
  --
  -- control vector = mode0 model start /x_busy dma_busy
  --
  -- idle_x
  IF (arb_state = "000") THEN
    IF (xbus OR both) THEN
      control_vec <= "00000";
      arb_state <= "010";
    ELSEIF dma_req THEN

  END IF;
END IF;
END IF;
END PROCESS moore;
END ARCHITECTURE arch_arb;
arb_state <= "011";
control_vec <= "00011";
ELSE
arb_state <= "000";
control_vec <= "00010";
END IF;
-- idle_d
ELSIF (arb_state = "001") THEN
  IF (xbus) THEN
    arb_state <= "010";
control_vec <= "00000";
  ELSIF (dma_req OR both) THEN
    arb_state <= "011";
control_vec <= "00011";
  ELSIF none THEN
    arb_state <= "001";
control_vec <= "00010";
  ELSE
    arb_state <= "000";
control_vec <= "00010";
  END IF;
-- cs_x
ELSIF (arb_state = "010") THEN
  IF stay THEN
    arb_state <= "010";
  IF xscsi THEN
    control_vec <= "00100";
  ELSIF xram THEN
    control_vec <= "01100";
  ELSIF dmamode THEN
    control_vec <= "10100";
  ELSE
    control_vec <= "00010";
  END IF;
ELSIF release THEN
  arb_state <= "100";
control_vec <= "00010";
ELSE
  arb_state <= "000";
control_vec <= "00010";
END IF;
-- cs_d
ELSIF (arb_state = "011") THEN
  IF stay THEN
    arb_state <= "011";
    control_vec <= "11111";
  ELSE
    arb_state <= "000";
    control_vec <= "00010";
  END IF;
END IF;

-- arb_wait
ELSIF (arb_state = "100") THEN
  IF not_done THEN
    arb_state <= "100";
    control_vec <= "00010";
  ELSE
    arb_state <= "000";
    control_vec <= "00010";
  END IF;
END IF;
END IF;
END PROCESS;

-- Logic Condensation
condense: PROCESS (sram_csl, control_vec(O))
BEGIN
  IF (sram_csl='0' AND control_vec(O)='1') THEN
    dma_bhe <= '0';
  ELSE
    dma_bhe <= '1';
  END IF;
END PROCESS;
END arch_arb;

A.2 Core State Machine PAL VHDL Listing

ENTITY core_state IS PORT (
  -- Arbitrator inputs/output
  clk, reset: IN BIT;
  mode0, mode1, start: IN BIT;
  xwrite, dma_dir: IN BIT;
  clk16: INOUT BIT;
  control_vec: INOUT x01z_vector(2 DOWNTO 0);
  state: INOUT x01z_vector(3 DOWNTO 0));
ATTRIBUTE pin_numbers OF core_state: ENTITY IS
  "clk:1 " &
  "mode0:2 " &
  "mode1:3 " &
  "reset:4 " &
  "start:5 " &
  "xwrite:6 " &
  "dma_dir:7 " &
  "clk16:23 " &
  "control_vec(2):22 " &
  "control_vec(1):21 " &
  "control_vec(0):20 " &
  "state(0):19 " &
  "state(1):18 " &
  "state(2):17 " &
  "state(3): 16 ";
END core_state;

USE WORK.rtlpkg.ALL;
USE WORK.int_math.ALL;

ARCHITECTURE arch_state OF core_state IS
  SUBTYPE state_var IS x01z_vector(3 DOWNTO 0);
  CONSTANT idle:state_var:="0000";
  CONSTANT waO:state_var:="0001";
  CONSTANT wbO:state_var:="0010";
  CONSTANT wa1:state_var:="0011";
  CONSTANT wb1:state_var:="0100";
  CONSTANT wa2:state_var:="0101";
  CONSTANT wb2:state_var:="0110";
  CONSTANT daO:state_var:="1000";
  CONSTANT da1:state_var:="1001";
  CONSTANT db1:state_var:="1010";
  CONSTANT rO:state_var:="1011";
  CONSTANT ral:state_var:="1100";
  CONSTANT rbl :state_var:="1101";
  SUBTYPE control_var IS x01z_vector(2 DOWNTO 0);
  CONSTANT wait_state:control_var:="011";
  CONSTANT x_ack:control_var:="001";
  CONSTANT x_ack_done:control_var:="101";
  CONSTANT d_ack:control_var:="010";
  CONSTANT done:control_var:="11";
BEGIN
  moore: PROCESS (clk, reset)
  VARIABLE dma_cycle, x_cycle: BOOLEAN;
  VARIABLE xscsi_r, xscsi_w, xsram: BOOLEAN;
  VARIABLE sram_scsi, scsi_sram, dma_mode: BOOLEAN;
BEGIN
  IF reset='1' THEN
    state <= idle;
    control_vec <= wait_state;
  ELSIF (clk'EVENT AND clk='1') THEN
    xscsi_r := (((mode0='0' AND mode1='0')AND clk16='0')AND start='1')AND xwrite='0');
    xscsi_w := (((mode0='0' AND mode1='0')AND clk16='0')AND start='1')AND xwrite='1');
    sram_scsi := (((mode0='1' AND mode1='1')AND start='1')AND dma_dir='0');
    scsi_sram := (((mode0='1' AND mode1='1')AND start='1')AND dma_dir='1');
    xsram := (((mode0='0' AND mode1='1')AND clk16='1')AND start='1');
    dma_mode := (((mode0='0' AND mode1='1')AND clk16='1')AND start='1');
  -- idle
  IF (state = idle) THEN
    IF xscsi_r THEN
      state <= wa0;
      control_vec <= wait_state;
    ELSIF xscsi_w THEN
      state <= wa2;
      control_vec <= wait_state;
    ELSIF sram_scsi THEN
      state <= db0;
      control_vec <= d_ack;
    ELSIF scsi_sram THEN
      state <= da0;
      control_vec <= d_ack;
    ELSIF xsram THEN
      state <= da1;
      control_vec <= wait_state;
    ELSIF dma_mode THEN
      state <= da1;
      control_vec <= wait_state;
    ELSE
      state <= idle;
  END IF;
control_vec <= wait_state;
END IF;

-- wa0
ELSIF (state = wa0) THEN
state <= wb0;
control_vec <= wait_state;

-- wb0
ELSIF (state = wb0) THEN
state <= wa1;
control_vec <= wait_state;

-- wa1
ELSIF (state = wa1) THEN
state <= wb1;
control_vec <= wait_state;

-- wb1
ELSIF (state = wb1) THEN
state <= wa2;
control_vec <= wait_state;

-- wa2
ELSIF (state = wa2) THEN
state <= wb2;
control_vec <= wait_state;

-- wb2
ELSIF (state = wb2) THEN
state <= da1;
control_vec <= wait_state;

-- da0:
ELSIF (state = da0) THEN
state <= db0;
control_vec <= d_ack;

-- db0:
ELSIF (state = db0) THEN
state <= da1;
control_vec <= d_ack;

-- da1
ELSIF (state = da1) THEN
IF (x_cycle) THEN
control_vec <= x_ack;
ELSE
control_vec <= wait_state;
END IF;
state <= db1;

-- db1
ELSIF (state = db1) THEN
  IF (x_cycle) THEN
    state <= ra1;
    control_vec <= x_ack_done;
  ELSIF (dma_cycle) THEN
    state <= r0;
    control_vec <= done;
  END IF;
-- r0
ELSIF (state = r0) THEN
  state <= ra1;
  control_vec <= done;
-- ra1
ELSIF (state = ra1) THEN
  state <= rb1;
  control_vec <= done;
-- rb1
ELSIF (state = rb1) THEN
  state <= idle;
  control_vec <= wait_state;
END IF;
END IF;
END PROCESS;

-- Clock synchronization for i960 cycles
clock16: PROCESS (clk, reset)
BEGIN
  IF (reset = '1') THEN
    clk16 <= '0';
  ELSIF (clk'EVENT AND clk='1') THEN
    clk16 <= NOT(clk16);
  END IF;
END PROCESS;
END arch_state;

A.3 Controller PAL VHDL Listing

ENTITY controller IS PORT (  
  -- Controller inputs
  clk, xwrite : IN BIT;
  dma_busy, start: IN BIT;
  xbe: IN BIT_VECTOR(1 DOWNTO 0);
  mode: IN BIT_VECTOR(0 TO 1);
);
state_vec: IN BIT_VECTOR(3 DOWNTO 0);
-- Control outputs
sram_cs: INOUT BIT_VECTOR(1 DOWNTO 0);
dma_dir, dma_be: INOUT BIT;
we, oe, scsi_cs: OUT BIT;
dma_a1_out: INOUT 'x01z;
dma_count, dma_reset: OUT BIT);
ATTRIBUTE pin_numbers OF controller: ENTITY IS
  "state_vec(0):2 " &
  "state_vec(1):3 " &
  "state_vec(2):4 " &
  "state_vec(3):5 " &
  "xbe(0):6 " &
  "xbe(1):7 " &
  "xwrite:8 " &
  "mode(0):9 " &
  "mode(1):10 " &
  "dma_busy:11 " &
  "start:13 " &
  "sram_cs(0):23 " &
  "sram_cs(1):22 " &
  "we:21 " &
  "oe:20 " &
  "scsi_cs:19 " &
  "dma_count:18 " &
  "dma_reset:17 " &
  "dma_a1_out:16 " &
  "dma_dir:15 " &
  "dma_be:14 ";
END controller;

USE WORK.rtlpkg.ALL;
USE WORK.int_math.ALL;

ARCHITECTURE arch_control OF controller IS
  SUBTYPE state_var IS BIT_VECTOR(3 DOWNTO 0);
  CONSTANT idle:state_var:="0000";
  CONSTANT wa0:state_var:="0001";
  CONSTANT wb0:state_var:="0010";
  CONSTANT wa1:state_var:="0011";
  CONSTANT wb1:state_var:="0100";
  CONSTANT wa2:state_var:="0101";
  CONSTANT wb2:state_var:="0110";
CONSTANT da0:state_var:="0111 ";
CONSTANT db0:state_var:="1000 ";
CONSTANT da1:state_var:="1001 ";
CONSTANT db1:state_var:="1010 ";
CONSTANT r0:state_var:="1011 ";
CONSTANT ral:state_var:="1100 ";
CONSTANT rbl:state_var:="1101 ";

SIGNAL dma_a1: BIT;
BEGIN
  equ_seg: PROCESS (clk)
  VARIABLE xscsi, xscsi_r, xscsi_w: BOOLEAN;
  VARIABLE ssram, ssram_r, ssram_w: BOOLEAN;
  VARIABLE xsram, xsram_r, xsram_w: BOOLEAN;
  VARIABLE dma_set: BOOLEAN;
  BEGIN
    IF (clk'EVENT AND clk='!') THEN

    -- Boolean expression to simplify if statements in the case statements
    _xscsi := (mode="00" AND start='l');
    xscsi_r := ((mode="00" AND start='!') AND xwrite='O');
    xscsi_w := ((mode="00" AND start='l') AND xwrite='l');
    ssram := (mode="11" AND start='!');
    ssram_r := ((mode="11" AND start='l') AND dma_dir='O');
    ssram_w := ((mode="11" AND start='l') AND dma_dir='l');
    xsram := (mode="01" AND start='l');
    xsram_r := ((mode="01" AND start='l') AND xwrite='O');
    xsram_w := ((mode="01" AND start='l') AND xwrite='l');
    dma_set := (mode="10" AND start='l');

    -- DMA counter reset and DMA transfer direction
    IF (state_vec=idle AND dma_set) THEN
      dma_reset <= '1';
      dma_dir <= xwrite;
      dma_a1 <= '0';
      dma_be <= '0';
    ELSE
      dma_reset <= '0';
      dma_dir <= dma_dir;
    IF (state_vec=r0 AND mode="11") THEN
      dma_be <= NOT(dma_be);
      dma_a1 <= ((dma_be AND NOT(dma_a1)) OR (NOT(dma_be)
      AND dma_a1));
    ELSE

    END PROCESS;
  ELSE

  END IF;
END;

51.
dma_al <= dma_al;
dma_be <= dma_be;
END IF;
END IF;
-- DMA count pulse
IF (((state_vec=rO AND mode="11")AND dma_al='l')AND dma_be='1') THEN
dma_count <= '1';
ELSE
dma_count <= '0';
END IF;
-- SCSI chip select
IF (NOT((state_vec=dbl OR state_vec=rO)OR state_vec=ral) AND xscsi) THEN
srsi_cs <= '0';
ELSE
srsi_cs <= '1';
END IF;
-- SRAM chip select
IF ((ssram OR xsram)AND NOT((state_vec=dbl OR state_vec=rO). OR state_vec=ra1)) THEN
sram_cs(O) <= NOT(((NOT(dma_be) AND mode(O))AND mode(!)) OR ((NOT(xbe(O)) AND NOT(mode(O))) AND mode(1)));
sram_cs(1) <= NOT(((_dma_be AND mode(O))AND mode(1)) OR ((NOT(xbe(1)) AND NOT(mode(O))) AND mode(1)));
ELSE
sram_cs(O) <= '1';
sram_cs(1) <= '1';
END IF;
-- Read/Write for SRAM/SCSI
IF (NOT(((state_vec=dbl AND state_vec=rO)AND state_vec=ra1)) AND ((xsram OR ssram)OR xscsi)) THEN
oe <= NOT(((xwrite AND NOT(mode(O))AND NOT(mode(1))) OR ((NOT(xwrite) AND NOT(mode(O)))AND mode(1)) OR ((NOT(dma_dir)AND mode(O))AND mode(1))));
we <= NOT(((NOT(xwrite) AND NOT(mode(O)))AND NOT(mode(1)))) OR ((xwrite AND NOT(mode(O)))AND mode(1))OR ((dma_dir AND mode(O))AND mode(1)));
ELSE
oe <= '1';
we <= '1';
END IF;
END IF;
END PROCESS;
52
tri: triout PORT MAP (dma_a1, dma_busy, dma_al_out);

END arch_control;

A.4 DMA Counter PAL VHDL Listing

ENTITY dmacnt IS PORT (  
    clk, do_count, reset, busy: IN BIT;  
    count_out: INOUT x01z_vector(9 DOWNTO 0));  
ATTRIBUTE pin_numbers OF dmacnt:ENTITY IS  
    "do_count:2" &  
    "reset:3 " &  
    "busy:4 " &  
    "count_out(1):23 " &  
    "count_out(3):22 " &  
    "count_out(5):21 " &  
    "count_out(7):20 " &  
    "count_out(9):19 " &  
    "count_out(8):18 " &  
    "count_out(6):17 " &  
    "count_out(4):16 " &  
    "count_out(2):15 " &  
    "count_out(0):14 ";
END dmacnt;
USE WORK.rtlpkg.ALL;
USE WORK.int_math.ALL;
ARCHITECTURE archdmacnt OF dmacnt IS
SIGNAL count: BIT_VECTOR(9 DOWNTO 0);
BEGIN
  counter: PROCESS (clk)
  BEGIN
    IF (clk'EVENT AND clk='1') THEN
      IF reset = '1' THEN
        count <= "0000000000";
      ELSE
        IF do_count = '0' THEN
          count <= count;
        ELSE
          count <= count+1;
        END IF;
      END IF;
    END IF;
  END PROCES;
END IF;
END PROCESS counter;
tstates: FOR i IN 9 DOWNTO 0 GENERATE
  tri: triout PORT MAP (count(i), busy, count_out(i));
END GENERATE;
END archdmacnt;
APPENDIX B
XBUS TIMING DIAGRAM

The following timing diagram relates the 80960 processor bus to the external bus of the EV80960SX board.

B.1 XBUS Read Timing Diagram

```
<table>
<thead>
<tr>
<th></th>
<th>T_a</th>
<th>T_w</th>
<th>T_d</th>
<th>T_r</th>
<th>T_a</th>
</tr>
</thead>
</table>
CLK16   |
|   | Φ_a | Φ_b | Φ_a | Φ_b | Φ_a |
XCLK    |
| *XCSO |
| XA[23:0] |
| XWRITE |
| DATA[15:0] |
| *XREADY |
```
APPENDIX C

EV80960SX LCD ADAPTER VHDL SOURCE

The controller for the LCD adapter consists of the modules that follow. The VHDL code was generated using System Architect from Mentor Graphics.

C.1 Clock Generator Module VHDL Listing

```vhdl
LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY clk_gen IS
  PORT(
    clk : IN STD_LOGIC;
    reset : IN STD_LOGIC;
    CP1 : OUT STD_LOGIC;
    CP2 : OUT STD_LOGIC;
    S : OUT STD_LOGIC;
  );
END clk_gen;

ARCHITECTURE rtl OF clk_gen IS
  SIGNAL pulse : STD_LOGIC;
  SIGNAL pulse_counter : pulse_range;
  SIGNAL cp1_counter : cp1_range;
  SIGNAL cp2_counter : cp2_range;

BEGIN
  pulse_count : PROCESS ( clk, reset )
  BEGIN
    IF (reset = '1') THEN
      pulse_counter <= To_stdlogicvector(0,5);
    END IF;
  END PROCESS;
```

56
ELSIF (clk'EVENT AND clk = '1') THEN
  IF (pulse_counter < To_stdlogicvector(21,5)) THEN
    pulse_counter <= pulse_counter + To_stdlogicvector(1,5);
  ELSE
    pulse_counter <= To_stdlogicvector(0,5);
  END IF;
END IF;
END PROCESS pulse_count;

pulse_gen : PROCESS (clk, reset)
BEGIN
  IF (reset = '1') THEN
    pulse <= '0';
  ELSIF (clk'EVENT AND clk = '1') THEN
    IF (pulse_counter < To_stdlogicvector(17,5)) THEN
      pulse <= '0';
    ELSE
      pulse <= '1';
    END IF;
  END IF;
END PROCESS pulse_gen;

CP2_count : PROCESS (pulse, reset)
BEGIN
  IF (reset = '1') THEN
    cp2_counter <= To_stdlogicvector(0,7);
  ELSIF (pulse'EVENT AND pulse='1') THEN
    IF (cp2_counter = To_stdlogicvector(81,7)) THEN
      cp2_counter <= To_stdlogicvector(1,7);  
    ELSE
      cp2_counter <= cp2_counter + To_stdlogicvector(1,7);
    END IF;
  END IF;
END PROCESS CP2_count;

CP_gen : PROCESS (clk, reset)
BEGIN
  IF (reset = '1') THEN
    CP2 <= 'O';
    CP1 <= '0';
  ELSIF (clk'EVENT AND clk='1') THEN
    IF (cp2_counter = To_stdlogicvector(81,7)) THEN
      CP2 <= '0';
    ELSE
      CP2 <= '1';
    END IF;
  END IF;
END PROCESS CP_gen;
CPl <= pulse;
ELSE
  CP2 <= pulse;
  CP1 <= '0';
END IF;
END IF;
END PROCESS CP_gen;

CP1_count: PROCESS ( pulse, reset)
BEGIN
  IF (reset = '1') THEN
    cp1_counter <= To_stdlogicvector(0,15);
  ELSIF (pulse'EVENT AND pulse='!') THEN
    IF (cp1_counter = To_stdlogicvector(19440,15)) THEN
      cp1_counter <= To_stdlogicvector(1,15);
    ELSE
      cp1_counter <= cp1_counter + To_stdlogicvector(1,15);
    END IF;
  END IF;
END IF;
END PROCESS CP1_count;

S_gen: PROCESS ( clk, reset)
BEGIN
  IF (reset = '1') THEN
    S<='0';
  ELSIF (clk'EVENT AND clk='!') THEN
    IF (cp1_counter < To_stdlogicvector(19440,15)) THEN
      S<='0';
    ELSE
      S<='1';
    END IF;
  END IF;
END IF;
END PROCESS S_gen;

END rtl;

C.2 Priority Encoder Module VHDL Listing

-- Component : priority_encoder

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY priority_encoder IS
  PORT (
    CP2 : IN STD_LOGIC;
    reset : IN STD_LOGIC;
    refresh_en : OUT STD_LOGIC;
    xbus_en : OUT STD_LOGIC
  );
END priority_encoder;

ARCHITECTURE rtl OF priority_encoder IS
  SIGNAL CP2_counter : pulse_range;
BEGIN

  vhdl_priority_encoder : PROCESS (CP2,
    reset,
    CP2_counter
  )
  BEGIN
    IF (reset = '1') THEN
      CP2_counter <= To_stdlogicvector(0,5);
      refresh_en <= '0';
      xbus_en <= '0';
    ELSIF (CP2'EVENT AND CP2 = '1') THEN
      IF (CP2_counter = To_stdlogicvector(20,5)) THEN
        CP2_counter <= To_stdlogicvector(1,5);
      ELSE
        CP2_counter <= CP2_counter + To_stdlogicvector(1,5);
      END IF;
    END IF;
    IF ((CP2_counter < To_stdlogicvector(14,5)) AND (CP2_counter
      = To_stdlogicvector(1,5))) THEN
      xbus_en <= '1';
      refresh_en <= '0';
    ELSIF (CP2_counter = To_stdlogicvector(16,5)) THEN
  END PROCESS vhdl_priority_encoder;

END rtl;

xbus_en <= '0';
  refresh_en <= '1';
ELSE
  xbus_en <= '0';
  refresh_en <= '0';
END IF;
END PROCESS vhdl_priority_encoder;
END rtl;

C.3 Output Buffer Module VHDL Listing

-- Component : buffers
LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY buffers IS
  PORT (rt_en : IN STD_LOGIC;
        refresh_cas : IN STD_LOGIC;
        refresh_en : IN STD_LOGIC;
        refresh_ras : IN STD_LOGIC;
        r_cas : IN STD_LOGIC;
        r_dsf : IN STD_LOGIC;
        r_ras : IN STD_LOGIC;
        r_trg : IN STD_LOGIC;
        r_vramaddr : IN byte;
        r_w : IN STD_LOGIC;
        xbus_en : IN STD_LOGIC;
        x_cas : IN STD_LOGIC;
        x_done : IN STD_LOGIC;
        x_dsf : IN STD_LOGIC;
        x_ras : IN STD_LOGIC;
        x_trg : IN STD_LOGIC;
        x_vramaddr : IN byte;
        x_w : IN STD_LOGIC;
        CAS_bar : OUT STD_LOGIC BUS;
        Done : OUT STD_LOGIC BUS;
        DSF : OUT STD_LOGIC BUS;
ARCHITECTURE spec OF buffers IS

BEGIN

x_addr_blk: BLOCK (xbus_en = '1')
BEGIN
  VRAMAddress <= GUARDED STD_LOGIC_VECTOR(x_vramaddr);
END BLOCK x_addr_blk;

r_addr_blk: BLOCK (rt_en = '1')
BEGIN
  VRAMAddress <= GUARDED STD_LOGIC_VECTOR(r_vramaddr);
END BLOCK r_addr_blk;

x_ras_blk: BLOCK (xbus_en = '1')
BEGIN
  RAS_bar <= GUARDED STD_LOGIC(x_ras);
END BLOCK x_ras_blk;

x_cas_blk: BLOCK (xbus_en = '1')
BEGIN
  CAS_bar <= GUARDED STD_LOGIC(x_cas);
END BLOCK x_cas_blk;

refresh_ras_blk: BLOCK (refresh_en = '1')
BEGIN
  RAS_bar <= GUARDED STD_LOGIC(refresh_ras);
END BLOCK refresh_ras_blk;

refresh_cas_blk: BLOCK (refresh_en = '1')
BEGIN
  CAS_bar <= GUARDED STD_LOGIC(refresh_cas);
END BLOCK refresh_cas_blk;

r_ras_blk: BLOCK (rt_en = '1')
BEGIN

61
RAS_bar <= GUARDED STD_LOGIC(r_ras);
END BLOCK r_ras_blk;

r_cas_blk: BLOCK (rt_en = '1')
BEGIN
  CAS_bar <= GUARDED STD_LOGIC(r_cas);
END BLOCK r_cas_blk;

x_trg_blk: BLOCK (xbus_en = '1')
BEGIN
  TRG_bar <= GUARDED STD_LOGIC(x_trg);
END BLOCK x_trg_blk;

r_trg_blk: BLOCK (rt_en = '1')
BEGIN
  TRG_bar <= GUARDED STD_LOGIC(r_trg);
END BLOCK r_trg_blk;

done_blk: BLOCK (xbus_en = '1')
BEGIN
  Done <= GUARDED STD_LOGIC(x_done);
END BLOCK done_blk;

x_w_blk: BLOCK (xbus_en = '1')
BEGIN
  W_bar <= GUARDED STD_LOGIC(x_w);
END BLOCK x_w_blk;

r_w_blk: BLOCK (rt_en = '1')
BEGIN
  W_bar <= GUARDED STD_LOGIC(r_w);
END BLOCK r_w_blk;

x_dsf_blk: BLOCK (xbus_en = '1')
BEGIN
  DSF <= GUARDED STD_LOGIC(x_dsf);
END BLOCK x_dsf_blk;

r_dsf_blk: BLOCK (rt_en = '1')
BEGIN
  DSF <= GUARDED STD_LOGIC(r_dsf);
END BLOCK r_dsf_blk;

END spec;
C.4 Refresh Module VHDL Listing

-- Component : vramrefresh

LIBRARY arithmetic ;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY vramrefresh IS
  PORT ( 
    clk : IN STD_LOGIC;
    refresh_en : IN STD_LOGIC;
    reset : IN STD_LOGIC;
    refresh_cas : OUT STD_LOGIC;
    refresh_ras : OUT STD_LOGIC
  );
END vramrefresh ;

ARCHITECTURE state_machine OF vramrefresh IS
  TYPE vramrefresh_state_type IS ( 
    start_state,
    cas_on,
    cas_ras_on,
    ras_on1,
    ras_on2,
    ras_on3,
    ras_on4,
    idle
  );

-- SDS Defined State Signals
SIGNAL current_state : vramrefresh_state_type := start_state ;
SIGNAL next_state : vramrefresh_state_type := start_state ;
BEGIN

-----------------------------------------------
  clocked : PROCESS ( 
    clk,
    refresh_en,
BEGIN
  IF (reset = '1') THEN
    current_state <= start_state;
  ELSIF (clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0') THEN
    current_state <= next_state;
  END IF;
END PROCESS clocked;

set_next_state : PROCESS (
  current_state,
  clk,
  refresh_en,
  reset
) BEGIN
  next_state <= current_state;
  CASE current_state IS
    WHEN start_state =>
      IF (refresh_en='1') THEN
        next_state <= cas_on;
      END IF;
    WHEN cas_on =>
      IF (TRUE) THEN
        next_state <= cas_ras_on;
      END IF;
    WHEN cas_ras_on =>
      IF (TRUE) THEN
        next_state <= ras_on1;
      END IF;
    WHEN ras_on1 =>
      IF (TRUE) THEN
        next_state <= ras_on2;
      END IF;
  END CASE;
END PROCESS set_next_state;
WHEN ras_on2 =>
  IF ( TRUE ) THEN
    next_state <= ras_on3;
  END IF;

WHEN ras_on3 =>
  IF ( TRUE ) THEN
    next_state <= ras_on4;
  END IF;

WHEN ras_on4 =>
  IF ( TRUE ) THEN
    next_state <= idle;
  END IF;

WHEN idle =>
  IF ( refresh_en = '0' ) THEN
    next_state <= start_state;
  END IF;

WHEN OTHERS =>
  NULL;
END CASE;
END PROCESS set_next_state;

unclocked : PROCESS ( clk, refresh_en, reset )
BEGIN
  IF ( reset = '1' ) THEN
    -- Start State Actions
    refresh_ras <= '1';
    refresh_cas <= '1';
  ELSIF ( clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0' ) THEN
    -- Default Actions
    refresh_cas <= '1';
    refresh_ras <= '1';
END PROCESS;
WHEN ras_on2 =>
  IF ( TRUE ) THEN
    next_state <= ras_on3;
  END IF;

WHEN ras_on3 =>
  IF ( TRUE ) THEN
    next_state <= ras_on4;
  END IF;

WHEN ras_on4 =>
  IF ( TRUE ) THEN
    next_state <= idle;
  END IF;

WHEN idle =>
  IF ( refresh_en = '0' ) THEN
    next_state <= start_state;
  END IF;

WHEN OTHERS =>
  NULL;
END CASE;

END PROCESS set_next_state ;

BEGIN
  IF ( reset = '1' ) THEN
    -- Start State Actions
    refresh_ras <= '1';
    refresh_cas <= '1';
  ELSIF ( clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0' ) THEN
    -- Default Actions
    refresh_cas <= '1';
    refresh_ras <= '1';
-- State Actions
CASE current_state IS
WHEN start_state =>
  refresh_ras <= '1';
  refresh_cas <= '1';
WHEN OTHERS =>
  NULL;
END CASE;

-- Transition Actions
CASE current_state IS
WHEN start_state =>
  IF ( refresh_en='1' ) THEN
    refresh_cas <= '0';
  END IF;

WHEN cas_on =>
  IF ( TRUE ) THEN
    refresh_ras <= '0';
    refresh_cas <= '0';
  END IF;

WHEN cas_ras_on =>
  IF ( TRUE ) THEN
    refresh_ras <= '0';
  END IF;

WHEN ras_on1 =>
  IF ( TRUE ) THEN
    refresh_ras <= '0';
  END IF;

WHEN ras_on2 =>
  IF ( TRUE ) THEN
    refresh_ras <= '0';
  END IF;

WHEN ras_on3 =>
  IF ( TRUE ) THEN
    refresh_ras <= '0';
  END IF;

WHEN OTHERS =>

C.5 Row Transfer Module VHDL Listing:

-- Component : row_state_machine

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY row_state_machine IS
PORT(
    clk : IN STD_LOGIC;
    CP1 : IN STD_LOGIC;
    reset : IN STD_LOGIC;
    row_en : OUT STD_LOGIC;
    rt_en : OUT STD_LOGIC;
    r_cas : OUT STD_LOGIC;
    r_dsf : OUT STD_LOGIC;
    r_ras : OUT STD_LOGIC;
    r_trg : OUT STD_LOGIC;
    r_w : OUT STD_LOGIC;
);
END row_state_machine;

ARCHITECTURE state_machine OF row_state_machine IS
TYPE row_state_machine_state_type IS
    (start_state,
    ras_lo,
    wait0,
    cas_lo,
    wait1,
    wait2,
    ras_cas_hi
-- SDS Defined State Signals
SIGNAL current_state : row_state_machine_state_type := start_state;
SIGNAL next_state : row_state_machine_state_type := start_state;
BEGIN

-----------------------------------------------
clocked : PROCESS (
    clk,
    CP1,
    reset
)
-----------------------------------------------
BEGIN
IF (reset = '1') THEN
    current_state <= start_state;
ELSIF (clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0') THEN
    current_state <= next_state;
END IF;
END PROCESS clocked;

-----------------------------------------------
set_next_state : PROCESS (
    current_state,
    clk,
    CP1,
    reset
)
-----------------------------------------------
BEGIN
next_state <= current_state;
CASE current_state IS
WHEN start_state =>
    IF (CP1 = '1') THEN
        next_state <= ras_lo;
    END IF;

WHEN ras_lo =>
    IF (TRUE) THEN
        next_state <= wait0;
END PROCESS set_next_state;

-----------------------------------------------
END IF;

WHEN wait0 =>
  IF ( TRUE ) THEN
    next_state <= cas_lo;
  END IF;

WHEN cas_lo =>
  IF ( TRUE ) THEN
    next_state <= wait1;
  END IF;

WHEN wait1 =>
  IF ( TRUE ) THEN
    next_state <= wait2;
  END IF;

WHEN wait2 =>
  IF ( TRUE ) THEN
    next_state <= ras_cas_hi;
  END IF;

WHEN ras_cas_hi =>
  IF ( CP1 = '0' ) THEN
    next_state <= start_state;
  END IF;

WHEN OTHERS =>
  NULL;
END CASE;
END PROCESS set_next_state;

unclocked : PROCESS ( clk, CP1, reset )
BEGIN
  IF ( reset = '1' ) THEN
    -- Start State Actions
ELSIF (clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0') THEN
-- Default Actions
r_trg <= '0';
r_ras <= '1';
r_cas <= '1';
r_dsf <= '0';
r_w <= '1';
row_en <= '0';
rt_en <= '0';

-- State Actions
CASE current_state IS
  WHEN start_state =>
    r_trg <= '1';
r_ras <= '1';
r_cas <= '1';
r_dsf <= '0';
r_w <= '1';
row_en <= '0';
rt_en <= '0';
  WHEN OTHERS =>
    NULL;
END CASE;

-- Transition Actions
CASE current_state IS
  WHEN start_state =>
    IF (CPI = '1') THEN
      row_en <= '1';
r_trg <= '0';
r_cas <= '1';
r_dsf <= '0';
    END IF;
  WHEN ras_lo =>
    IF (TRUE) THEN
      r_ras <= '0';
    END IF;
END CASE;
```vhdl
row_en <= '1';
END IF;

WHEN wait0 =>
  IF ( TRUE ) THEN
    r_ras <= '0';
  END IF;

WHEN cas_lo =>
  IF ( TRUE ) THEN
    r_ras <= '0';
    r_cas <= '0';
  END IF;

WHEN wait1 =>
  IF ( TRUE ) THEN
    r_ras <= '0';
    r_cas <= '0';
    r_trg <= '1';
  END IF;

WHEN wait2 =>
  IF ( TRUE ) THEN
    r_ras <= '0';
    r_cas <= '0';
    r_trg <= '1';
  END IF;

WHEN ras_cas_hi =>
  IF ( CPI = '0' ) THEN
    rt_en <= '0';
    r_trg <= '1';
  END IF;

WHEN OTHERS =>
  NULL;
END CASE;

END IF;

END PROCESS unclocked;
END state_machine;
```
LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY row_address IS
  PORT(
    CP1 : IN STD_LOGIC;
    reset : IN STD_LOGIC;
    row_en : IN STD_LOGIC;
    r_vramaddr : OUT byte
  );
END row_address;

ARCHITECTURE spec OF row_address IS
  SIGNAL r_rowaddr : BYTE;
BEGIN

  vhdl_row_address : PROCESS (CP1, reset)
  BEGIN
    IF (reset = '1') THEN
      r_rowaddr <= To_stdlogicvector(1, 9);
    ELSIF (CP1'EVENT AND CP1 = '1') THEN
      IF (r_rowaddr = To_stdlogicvector(239, 9)) THEN
        r_rowaddr <= To_stdlogicvector(0, 9);
      ELSE
        r_rowaddr <= r_rowaddr + To_stdlogicvector(1, 9);
      END IF;
    END IF;
  END PROCESS vhdl_row_address;

  address_select : PROCESS (row_en, r_rowaddr)
  BEGIN

END ARCHITECTURE spec;
IF (row_en = '1') THEN
    r_vramaddr <= r_rowaddr;
ELSE
    r_vramaddr <= To_stdlogicvector( 0, 9);
END IF;
END PROCESS address_select;
END spec ;

C.6 XBUS Transfer Module VHDL Listing

LIBRARY arithmetic ;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY phase_gen IS
    PORT ( 
        clk : IN STD_LOGIC; 
        reset : IN STD_LOGIC; 
        phasea : OUT STD_LOGIC
    );
END phase_gen ;

ARCHITECTURE rtl OF phase_gen IS
    SIGNAL int_phase : STD_LOGIC;
BEGIN

vhdl_phase_gen : PROCESS ( 
    clk, 
    reset 
)
BEGIN
    IF (reset = '1') THEN 
        int_phase <= '0';
    ELSIF ( clk'EVENT AND clk = '1') THEN 
        int_phase <= NOT int_phase;

END PROCESS vhdl_phase_gen;

END rtl;
END IF;

END PROCESS vhdl_phase_gen;

phasea <= int_phase;
END rtl;

-- Component: xbus_state_machine

LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY xbus_state_machine IS
PORT(
  clk : IN STD_LOGIC;
  LCD : IN STD_LOGIC;
  phasea : IN STD_LOGIC;
  reset : IN STD_LOGIC;
  xbus_en : IN STD_LOGIC;
  XWrite : IN STD_LOGIC;
  addr_select : OUT STD_LOGIC;
  XB_ena : OUT STD_LOGIC;
  XReady : OUT STD_LOGIC;
  x_cas : OUT STD_LOGIC;
  x_done : OUT STD_LOGIC;
  x_dsf : OUT STD_LOGIC;
  x_ras : OUT STD_LOGIC;
  x_trg : OUT STD_LOGIC;
  x_w : OUT STD_LOGIC
);
END xbus_state_machine;

ARCHITECTURE state_machine OF xbus_state_machine IS
TYPE xbus_state_machine_state_type IS(
  start_state,
  XB_ena_active,
  w_active,
  cas_10,
  wait3,
  ...
-- SDS Defined State Signals.
SIGNAL current_state : xbus_state_machine_state_type := start_state;
SIGNAL next_state : xbus_state_machine_state_type := start_state;
BEGIN

---------------------------------------------------------------
clocked : PROCESS (clk,

---------------------------------------------------------------
clocked : PROCESS (clk,
BEGIN
IF (reset = 'I') THEN
  current_state <= start_state;
ELSIF (clk'EVENT AND clk = 'I' AND clk'LAST_VALUE = '0') THEN
  current_state <= next_state;
END IF;

END PROCESS clocked;

---------------------------------------------------------------
set_next_state : PROCESS (current_state,
BEGIN
next_state <= current_state;
CASE current_state IS
  WHEN start_state =>
    IF ( xbus_en = '1' AND LCD = '1' AND phasea = '1' ) THEN
      next_state <= XB_ena_active;
      END IF;
  WHEN XB_ena_active =>
    IF ( TRUE ) THEN
      next_state <= ras_lo;
      END IF;
  WHEN w_active =>
    IF ( TRUE ) THEN
      next_state <= cas_lo;
      END IF;
  WHEN cas_lo =>
    IF ( TRUE ) THEN
      next_state <= ready_lo;
      END IF;
  WHEN wait3 =>
    IF ( TRUE ) THEN
      next_state <= done;
      END IF;
  WHEN done =>
    IF ( TRUE ) THEN
      next_state <= idle;
      END IF;
  WHEN idle =>
    IF ( xbus_en = '0' ) THEN
      next_state <= start_state;
      END IF;
  WHEN ras_lo =>
    IF ( TRUE ) THEN
      next_state <= w_active;
      END IF;
  WHEN ready_lo =>
IF (TRUE) THEN
    next_state <= wait3;
END IF;

WHEN OTHERS =>
    NULL;
END CASE;

END PROCESS set_next_state;

------------------------------

uncelocked : PROCESS (
    clk,
    LCD,
    phasea,
    reset,
    xbus_en,
    XWrite
) 
BEGIN
IF (reset = '1') THEN
    -- Start State Actions
    XReady <= '1';
    x_done <= '0';
    x_ras <= '1';
    x_w <= '1';
    x_trg <= '1';
    x_dsfi <= '0';
    addr_select <= '0';
    XB_ena <= '1';
    x_cas <= '1';
ELSIF (clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0') THEN
    -- Default Actions
    XReady <= '1';
    x_done <= '0';
    x_ras <= '1';
    x_w <= '1';
    x_trg <= '1';
    x_dsfi <= '0';
    addr_select <= '0';
    XB_ena <= '0';
    x_cas <= '1';

------------------------------
-- State Actions
CASE current_state IS
WHEN start_state =>
  XReady <= '1';
  x_done <= '0';
  x_ras <= '1';
  x_w <= '1';
  x_trg <= '1';
  x_dsf <= '0';
  addr_select <= '0';
  XB_ena <= '1';
  x_cas <= '1';
WHEN OTHERS =>
  NULL;
END CASE;

-- Transition Actions
CASE current_state IS
WHEN start_state =>
  IF ( xbus_en = '1' AND LCD = '1' AND phasea = '1') THEN
    XB_ena <= '0';
  END IF;

WHEN XB_ena_active =>
  IF ( TRUE ) THEN
    x_ras <= '0';
  END IF;

WHEN w_active =>
  IF ( TRUE ) THEN
    x_trg <= XWrite;
    x_ras <= '0';
    x_w <= NOT XWrite;
    addr_select <= '1';
    x_cas <= '0';
  END IF;

WHEN cas_lo =>
  IF ( TRUE ) THEN
    x_ras <= '0';
    x_w <= NOT XWrite;
    x_cas <= '0';
x_trg <= XWrite;
addr_select <= '1';
XReady <= '0';
END IF;

WHEN wait3 =>
  IF ( TRUE ) THEN
    x_done <= '1';
    XReady <= '0';
    x_ras <= '0';
    x_cas <= '0';
    x_w <= NOT XWrite;
  END IF;

WHEN done =>
  IF ( TRUE ) THEN
    XB_ena <= '1';
  END IF;

WHEN idle =>
  IF ( xbus_en = '0' ) THEN
    XB_ena <= '1';
  END IF;

WHEN ras_lo =>
  IF ( TRUE ) THEN
    x_ras <= '0';
    x_w <= NOT XWrite;
    addr_select <= '1';
  END IF;

WHEN ready_lo =>
  IF ( TRUE ) THEN
    XReady <= '0';
    x_ras <= '0';
    x_cas <= '0';
    x_w <= NOT XWrite;
  END IF;

WHEN OTHERS =>
  NULL;
END CASE;
END IF;

END PROCESS unclocked;
END state_machine;

-- Component : xbus_address:
LIBRARY arithmetic;
USE arithmetic.std_logic_arith.ALL;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.lcd_types.ALL;

ENTITY xbus_address IS
PORT(
   addr_select : IN STD_LOGIC;
   Col_addr : IN byte;
   Row_addr : IN byte;
   x_vramaddr : OUT byte
);
END xbus_address;

ARCHITECTURE rtl OF xbus_address IS
BEGIN

vhdl_xbus_address : PROCESS (.addr_select, Col_addr, Row_addr)
BEGIN
CASE addr_select IS
WHEN '0' =>
   x_vramaddr <= Row_addr;
WHEN '1' =>
   x_vramaddr <= Col_addr;
WHEN OTHERS =>
   x_vramaddr <= "00000000";
END CASE;
END PROCESS vhdl_xbus_address;
END rtl;
APPENDIX D
ECE 311 MONITOR SOURCE CODE

The following source code listings are from the ECE 311 monitor program. The code is compiled and downloaded into the EV80960SX board to provide a user interface to the SCSI and LCD adapters.

D.1 ECE 311 Monitor DMA Buffer Routines

#ifndef SARAM
#define SARAM

#include <stdio.h>
#include <error.h>
#include <glob_311.h>

void disp_sram(struct cmd *command)
{
    int start, finish, i;
    volatile unsigned char *byte_data;

    start = command->arg[0];
    finish = command->arg[1];

    if (command->nargs != 2)
    {
        print_err(NUM_ARGS,command);
        return;
    }

    // Rest of the code goes here...
}
#endif
if (finish < start) {
    print_err(INV_RANGE,command);
    return;
}

if ((start < 0) || (start > 4095)) {
    print_err(INV_ARG1,command);
    return;
}

if ((finish < 0) || (finish > 4095)) {
    print_err(INV_ARG2,command);
    return;
}

start = command->arg[0] + SRAM_BASE_ADDR;
finish = command->arg[1] + SRAM_BASE_ADDR;

printf("n");fflush(stdout);
while (finish >= start) {
    printf("%8X: ",start);fflush(stdout);
    for (i=1;((i <= 16) && (finish >= start));i++) {
        if (i == 9) printf(":");fflush(stdout);
        byte_data = (volatile unsigned char *)start;
        printf("%2X ", *byte_data);fflush(stdout);
        start += sizeof(unsigned char);
    }
    printf("n\n");fflush(stdout);
}

fill_sram(command)
{
    struct cmd *command;
    
    int start, finish, fill, i;
    volatile unsigned char *byte_data;

    start = command->arg[0];
    finish = command->arg[1];
    fill = command->arg[2];

    if (command->nargs != 3) {
        print_err(NUM_ARGS,command);
        return;
    }
if (finish < start){
    print_err(INV_RANGE,command);
    return;
}

if ((start < 0) || (start > 4095)) {
    print_err(INV_ARG1,command);
    return;
}

if ((finish < 0) || (finish > 4095)) {
    print_err(INV_ARG2,command);
    return;
}

if ((fill < 0) || (fill > 255)) {
    print_err(INV_ARG3,command);
    return;
}

start = command->arg[0] + SRAM_BASE_ADDR;
finish = command->arg[1] + SRAM_BASE_ADDR;

while (finish >= start) {
    *(volatile unsigned char *)start = fill;
    start += sizeof(unsigned char);
}

mod_sram(command)
struct cmd *command;
{
    int addr;

    if (command->nargs != 2) {
        print_err(NUM_ARGS,command);
        return;
    }

    if ((command->arg[0] < 0) || (command->arg[0] > 4095)) {
        print_err(INV_ARG1,command);
        return;
    }

print_err(INV_ARG2, command);
return;
}

addr = SRAM_BASE_ADDR + command->arg[0];
*(volatile unsigned char *)addr = command->arg[1];

brutalize_sram()
{
    int start, finish, data, i, j;
    start = 0;
    finish = 4095;

    start += SRAM_BASE_ADDR;
    finish += SRAM_BASE_ADDR;

    while (finish >= start){
        for (i=0; i<256; i++) {
            *(volatile unsigned char *)start = i;
            data = *(volatile unsigned char *)start;
            if (i != data){
                printf("BOOM! %8X %d\n", start, i);
                fflush(stdout);
                return(0);
            }
        }
    }

    if (((start+1-SRAM_BASE_ADDR) % 256) == 0)
        printf("%d Bytes Tested\n", (start+1-SRAM_BASE_ADDR));
    fflush(stdout);
    start += sizeof(unsigned char);
}

for (j=0; j<256; j++){
    start = SRAM_BASE_ADDR;
    while (finish >= start){
        *(volatile unsigned char *)start = (start+j)&0xFF;
        start += sizeof(unsigned char);
    }
    start = SRAM_BASE_ADDR;
    while (finish >= start){
        data = *(volatile unsigned char *)start;
        if (((start+j)&0xff) != data){
            printf("BOOM! %8X %d\n", start, j);
            fflush(stdout);
        }
    }
}

84
D.2 ECE 311 Monitor VRAM Routines

/*
   Copyright (c) 1993, 1994, University of Illinois.
   */
/* Changes:
 * 5/10/94 Kevin Nickels
 *   Added version information
 * 11/20/94 Kevin Nickels
 *   Changed get* to get*_hd
 * 12/14/94 Kevin Nickels
 *   Fixed the annoying no-backspace bug... was using
 *   gets() instead of readline() to get filename.
 *   Changed nibble_in/nibble_out to the way we ask the
 *   students to do it. (see lite_bar.c)
 *   Changed hard limits on size of bitmap to dump
 *   (overflow protection) to use #defines
 *   MAX_LCD_WIDTH, MAX_LCD_HEIGHT (see glob_311.h)
 *   Only read in min(MAX_LCD_HEIGHT,height) lines of file.
 */

#include <stdio.h>
#include <error.h>
#include <errno.h>
#include <string.h>
#include <glob_311.h>

85
nibble_out (row, col, data)
    int row; int col; unsigned char data;
{
    volatile unsigned char *addr =
        (unsigned char *)(VRAM_BASE_ADDR + (row<<10) + (col<<1));
    *addr = 0xf & data;
}

int nibble_in (row, col)
    int row; int col; {
    volatile unsigned char *addr =
        (unsigned char *)(VRAM_BASE_ADDR + (row<<10) + (col<<1));
    return 0xf & *addr;
}

/*
 * Flip nibble -- so the picture looks right
 */
unsigned char flip_nibble (data)
    unsigned char data;
{
    data = data & 0xf;
    return ((data&0x1)<<3) |
        ((data&0x2)<<1) |
        ((data&0x4)>>1) |
        ((data&0x8)>>3) ;
}

static char buffer[1024];
static int ptr;
static int size;

struct icon {
    int width, height;
    unsigned char *data;
};
static struct icon icon;

int min(int a, int b)
{
    if (a<b) return a; else return b;
}
int get_dec_number_hd(FILE *f)
{
    char *p;
    if (fgets(buffer,sizeof(buffer),f)=='\0') return -1;
    if ((p = strchr(buffer,' '))==\0) return -1;
    if ((p = strchr(p+1,' ')) == \0) return -1;
    return atoi(p);
}

/* Paint bitmap from hard drive
 * 1) Get filename from user
 * 2) Open file
 * 3) Read Header
 * 4) Allocate memory for structure of nibbles (actually, chars).
 * 5) Read in nibbles (conversion from bytes->nibbles on the fly)
 * 6) Clear the screen and output nibbles
 * Note: ~ = bitwise complement
 * flip_nibble is used because we get the data in logical bit-order and
 * the LCD is in strange order (D3D2D1D0)(D7D6D5D4)(etc)
 */

paint_bitmap_from_hd() {
    FILE *f;
    char fn_buffer[256];
    int numBytes,numNibbles,widthBytes,widthNibbles,k;
    int r,c;

    /* Get Filename */
    printf("Enter filename: "); fflush(stdout);
    getline(fn_buffer, 256);
    printf("\n"); fflush(stdout);
    errno = \0;
    f=fopen(fn_buffer,"r");
    if ( f == \0 ) {
        printf("Open of %s for input failed: %s\n", fn_buffer, strerror(errno));
        fflush(stdout);
        return;
    }

    /* Read icon header and malloc memory */

/* First line - #define smiley_width 36 */
icon.width = get_dec_number_hd(f);

/* Second line - #define smiley_height 36*/
icon.height = min(get_dec_number_hd(f), MAX_LCD_HEIGHT);

/* Throw out the third line - static char smiley_bits[] = { */
widthBytes = ((icon.width + 7) / 8);
widthNibbles = (widthBytes) * 2;
umNibbles = widthNibbles * icon.height;
printf("This icon is %d wide and %d high; allocating %d Nibbles\n", icon.width, icon.height, numNibbles);fflush(stdout);
icon.data =
    (unsigned char *)malloc(numNibbles*sizeof(unsigned char));
if (icon.data == NULL) {
    printf("Could not malloc enough memory for icon. :\n\n");
    fflush(stdout);
    return;
}

/* Initialize get_hex_number_hd() and read in bytes */
size = ptr = 0;
for (r=0; r<icon.height; r++) {
    /* Print status updates */
    if ((icon.height-r) % 10 == 0) {
        printf("%d ",icon.height-r);
        fflush(stdout);
    }

    for (c=0; c<widthBytes; c++) {
        /* Grab the number from the file */
        k = (get_hex_number_hd(f) & 0xff);
        /* Store lower nibble */
        icon.data[r*widthNibbles+c*2] = k & 0xf;
        /* Store upper nibble */
        icon.data[r*widthNibbles+c*2+1] = (k & 0xf0) >> 4;
    }
}
printf("\n\n");fflush(stdout);

/* Erase VRAM */
for (r=0; r<512; r++)
    for (c=0; c<512; c++)
nibble_out(r, c, (unsigned char) 0xF);

/* Put icon to VRAM */
for (r = 0; r < min(icon.height, MAX_LCD_HEIGHT); r++)
    for (c = 0; c < min(widthNibbles, MAX_LCD_WIDTH); c++)
        .nibble_out(r, c, flip_nibble(~icon.data[r*widthNibbles+c]));

free(icon.data);
fclose(f);
}

disp_vram(command)
struct cmd *command;
{
    int row, col, startrow, startcol, finishrow, finishcol, i;
    volatile unsigned char data, fill;
    startrow = command->arg[0];
    startcol = command->arg[1];
    finishrow = command->arg[2];
    finishcol = command->arg[3];
    fill = (unsigned char) command->arg[4];

    if (command->nargs != 4)
        print_err(NUM_ARGS, command); return;

    if (finishrow < startrow || finishcol < startcol){
        print_err(INV_RANGE, command); return;
    }

    if ((startrow < 0) || (startrow > 0x1ff))
        print_err(INV_ARG1, command); return;

    if ((startcol < 0) || (startcol > 0x1ff))
        print_err(INV_ARG2, command); return;

    if ((finishrow < 0) || (finishrow > 0x1ff))
        print_err(INV_ARG3, command); return;

    if ((finishcol < 0) || (finishcol > 0x1ff))
        print_err(INV_ARG4, command); return;

    row = startrow;
    printf("Display %x %x to %x %x\n", startrow, startcol, finishrow, finishcol);
fflush(stdout);
while (row <= finishrow) {
    col = startcol;
    printf("\nAddress %8x (Row %2x) : ",
        VRAM_BASE_ADDR+(row<<8),row);
    fflush(stdout);
    for (i=1; (col <= finishcol); i++) {
        if (i%9==0) printf(" :: ");fflush(stdout);
        data:::; nibble_in(row,col);
        printf("%x ",0xf & data);fflush(stdout);
        col++;
    }
    row++;
}
printf("\n\n");fflush(stdout);

fill_vram(command)
struct cmd *command;
{
    int row, col, startrow,startcol, finishrow, finishcol, i;
    volatile unsigned char *data, fill;

    startrow = command->arg[0];
    startcol = command->arg[1];
    finishrow = command->arg[2];
    finishcol = command->arg[3];
    fill = (unsigned char) command->arg[4];

    if (command->nargs != 5){
        print_err(NUM_ARGS,command); return; }

    if (finishrow < startrow || finishcol < startcol){
        print_err(INV_RANGE,command); return; }

    if ((startrow < 0) || (startrow > 0xff)){
        print_err(INV_ARG1,command); return; }

    if ((startcol < 0) || (startcol > 0xff )){
        print_err(INV_ARG2,command); return; }

    if ((finishrow < 0) || (finishrow > 0xff)){
        print_err(INV_ARG3,command); return; }
if ((finishcol < 0) || (finishcol > 0x1ff)) {
    print_err(INV_ARG4,command); return; }

if (fill > 0xf) {
    print_err(INV_ARG5,command); return; }

row=startrow;
while (row <= finishrow) {
    col=startcol;
    while (col <= finishcol) {
        nibble_out(row,col,fill);
        col++;
    }
    row++;
}

mod_vram(command)
{
    struct cmd *command;
    {
        int row,col,addr,data,readback;
        if (command->nargs != 3) {
            print_err(NUM_ARGS,command); return;
        }

        if ((command->arg[0] < 0) || (command->arg[0] > 0x1ff)) {
            print_err(INV_ARG1,command); return;
        }

        if ((command->arg[1] < 0) || (command->arg[1] > 0x1ff)) {
            print_err(INV_ARG2,command); return;
        }

            print_err(INV_ARG3,command); return;
        }

        nibble_out(command->arg[0],command->arg[1],command->arg[2]);
        readback = nibble_in(command->arg[0],command->arg[1]);

        printf("Address %8x : (row %2x, column %2x) : modified to %x \n",
            addr,command->arg[0],command->arg[1],command->arg[2]);
        printf("Address %8x : (row %2x, column %2x) : read back as %x\n",
            addr,command->arg[0],command->arg[1],readback);
        fflush(stdout);
    }
}
D.3 ECE 311 Interrupt Routines

```
.text
.globl _xbusa_int_isr
.globl _mon_xbusa_int_isr
.globl _set_priority
.globl _user_isr
.globl _scsi_semaphore
.globl _scsi_message
.globl _scsi_num

.align 2
_set_priority:
   ldconst 64, r4
   addo sp, r4, sp
   stq g0, -64(sp)
```

All user interrupt handlers are placed here. Assembly language is required to insure that the global registers are not clobbered.

```
.set_priority(int arg):
   arg: is a priority level where 0 - lowest, 31 - highest
   This should not be necessary, but it is here if needed.
   According to the NINDY Documentation, NINDY runs at level 31. We assume the user program will inherit that value.
   Therefore, we have to lower it to see the interrupts. This may or may not be the case.
```

92
stq g4, -48(sp)
stq g8, -32(sp)
std g12, -16(sp)

ldconst 0x001f0000, gl
modpc g0, gl, g0

ldq -64(sp), g0
ldq -48(sp), g4
ldq -32(sp), g8
ldt -16(sp), g12
ret

/*************************************************************************/
/*
 * xbusa_int_isr()
 */
/*
 * This is the ISR for the SCSI chip.
 * This routine is used by the ISO-9660 filesystem.
 */

.align 2

_xbusa_int_isr:

ldconst 64, r4
addo sp, r4, sp

stq g0, -64(sp)
stq g4, -48(sp)
stq g8, -32(sp)
std g12, -16(sp)

/* Acknowledge Interrupt Controller */
ldconst 0x61, gl
lda 0x28000000, g2
stob g1, (g2)

/* Find out what interrupt actually occurred */
ldconst 0x17, g1
lda 0x30000000, g2
stob g1, (g2).

lda 0x30000002, g2
ldob (g2), g1
stob g1, _scsi_num
/* Read message passed */
ldconst 0x0f, g1
lda 0x30000000, g2
stob g1, (g2)
lda 0x30000002, g2
ldob (g2), g1
stob g1, _scsi_message.

/* Determine Phase */
ldconst 0x10, g1
lda 0x30000000, g2
stob g1, (g2):
lda 0x30000002, g2
ldob (g2), g1
stob g1, _scsi_phase

/* Release SCSI semaphore */
ldconst 0x00, g1
stob g1, _scsi_semaphore

ldq -64(sp), g0
ldq -48(sp), g4
ldq -32(sp), g8
ldt -16(sp), g12
ret

/* Used for the monitor to return only the interrupt number from SCSI Chip*/
.align 2
_mon_xbusa_int_isr:
ldconst 64, r4
addo sp, r4, sp

stq g0, -64(sp)
stq g4, -48(sp)
stq g8, -32(sp)
stt g12, -16(sp)

/* Acknowledge Interrupt Controller */
ldconst 0x61, g1
ida 0x28000000, g2
stob g1, (g2)
/* Find out what interrupt actually occurred */
ldconst 0x17,g1
lda 0x30000000, g2
stob g1, (g2)

lda 0x30000002, g2
ldob (g2), g1

lda fmt1, g0
call _printf

ldq -64(sp), g0
ldq -48(sp), g4
ldq -32(sp), g8
ldt -16(sp), g12
ret

.align 2

D.4 ECE 311 Monitor SCSI Routines

ifndef dis_scsi_reg
#define dis_scsi_reg(command)
    struct cmd *command;
{
    int scsi_reg, scsi_base;
#endif

/*
 * Header: scsi.c,v 1.195/01/04 12:05:27 greenlaw Exp
 */
/*
 * Copyright (c) 1993, 1994, University of Illinois
 */

#include <stdio.h>
#include <glob_311.h>
#include <error.h>

dis_scsi_reg(command)
    struct cmd *command;
{
scsi_reg = SCSI_BASE_ADDR + 2;
scsi_base = SCSI_BASE_ADDR;
*(volatile unsigned char *)scsi_base = command->arg[0];
printf("\n Reg %2X = %2X\n", command->arg[0],
*(volatile unsigned char *)scsi_reg); fflush(stdout);

mod_scsi_reg(command)
struct cmd *command;
{
    int scsi_reg, scsi_base;
    scsi_base = SCSI_BASE_ADDR;
    scsi_reg = SCSI_BASE_ADDR + 2;
    *(volatile unsigned char *)scsi_base = command->arg[0];
    *(volatile unsigned char *)scsi_reg = command->arg[1];
    printf("\n Reg %2X set to %2X\n", command->arg[0], command->arg[1]);
    fflush(stdout);
}

dma_init(command)
int command;
{
    int dma_base, scrap;
    dma_base = DMA_BASE_ADDR;
    if (command == DMA_WRITE)
        *(volatile unsigned char *)dma_base = 0x00;
    else /* DMA_READ */
        scrap = *(volatile unsigned char *)dma_base;
}

D.5 ECE 311 Kernel SCSI Device Driver Routines

 /**************************************************************************/
 /* Copyright (c) 1993, University of Illinois */
 /**************************************************************************/
 /*
 * $Header: sctl.c,v 1.1 95/01/04 12:07:06 greenlaw Exp $
 * $Revision: 1.1 $
 * $Log: sctl.c,v $
 * Revision 1.1 95/01/04 12:07:06 12:07:06 greenlaw (Jonathan Greenlaw)
 * Initial revision
 */
/*
 */
#include <stdio.h>
#include <scsi.h>

int scsi_num;
int scsi_message;
int scsi_semaphore;
int scsi_phase;

#define SCSI_BASE_ADDR 0x30000000
#define DMA_READ 0
#define DMA_WRITE 1

int scsi_init()
{
    volatile unsigned char *byte_data;
    scsi_semaphore = 1;
    scsi_base = SCSI_BASE_ADDR;
    scsi_reg = SCSI_BASE_ADDR + 2;

    fs_insert_intr();
    mod_intr_mask(0xfd);
    set_priority(0);

    /* Initialize SCSI ID and Soft reset*/
    *(volatile unsigned char *)scsi_base = 0x00;
    *(volatile unsigned char *)scsi_reg = 0x8F;
    *(volatile unsigned char *)scsi_base = 0x18;
    *(volatile unsigned char *)scsi_reg = 0x00;

    while (scsi_semaphore == 1)
    {
        eat_time(1);
    
    return 1;
}

int scsi_io(command)
struct scsi_log *command;
{
    volatile unsigned char *byte_data;
    int scsi_reg, scsi_base, i, scrap_reg, scrap_base;
}
scsi_base = SCSI_BASE_ADDR;
scsi_reg = SCSI_BASE_ADDR + 2;

scsi_semaphore = 1;

if ((command->cdb[0] == 0x15) || (command->cdb[0] == 0x55))
    dma_init(DMA_READ);
else
    dma_init(DMA_WRITE);

*(volatile unsigned char *)scsi_base = 0x00;
*(volatile unsigned char *)scsi_reg = command->cdb_length;

*(volatile unsigned char *)scsi_base = 0x01;
*(volatile unsigned char *)scsi_reg = 0x88;

*(volatile unsigned char *)scsi_base = 0x02;
*(volatile unsigned char *)scsi_reg = ((command->max_msecs) * 16)/80;

for (i=0; i < command->cdb_length; i++) {
    *(volatile unsigned char *)scsi_base = i + 3;
    *(volatile unsigned char *)scsi_reg = command->cdb[i];
}

*(volatile unsigned char *)scsi_base = 0x12;
*(volatile unsigned char *)scsi_reg = command->size[1];

*(volatile unsigned char *)scsi_base = 0x13;
*(volatile unsigned char *)scsi_reg = command->size[2];

*(volatile unsigned char *)scsi_base = 0x14;
*(volatile unsigned char *)scsi_reg = command->size[3];

*(volatile unsigned char *)scsi_base = 0x15;
if ((command->cdb[0] == 0x15) || (command->cdb[0] == 0x55))
    *(volatile unsigned char *)scsi_reg = (command->target_id);
else
    *(volatile unsigned char *)scsi_reg = (0x40 | command->target_id);

*(volatile unsigned char *)scsi_base = 0x18;
*(volatile unsigned char *)scsi_reg = 0x09;
while ( scsi_semaphore == 1 )
    eat_time(1);

*(volatile unsigned char *)scsi_base = 0x17;
scsi_num = *(volatile unsigned char *)scsi_reg;

while (scsi_num != 0x16 && scsi_num != 0){

    printf("nSCSI Problem. Status=\%2X Message=\%2X Phase=\%2X\n",)
    scsi_num, scsi_message, scsi_phase);
    printf("Trying to recover...\n");fflush(stdout);

    switch (scsi_num) {
    case 0x17:
        scsi_num = 0x16;
        break;
    case 0x4b:
        scsi_semaphore = 1;

        *(volatile unsigned char *)scsi_base = 0x10;
        *(volatile unsigned char *)scsi_reg = 0x41;

        *(volatile unsigned char *)scsi_base = 0x12;
        *(volatile unsigned char *)scsi_reg = 0;

        *(volatile unsigned char *)scsi_base = 0x13;
        *(volatile unsigned char *)scsi_reg = 0;

        *(volatile unsigned char *)scsi_base = 0x14;
        *(volatile unsigned char *)scsi_reg = 0;

        *(volatile unsigned char *)scsi_base = 0x18;
        *(volatile unsigned char *)scsi_reg = 0x09;

        while ( scsi_semaphore == 1 )
            eat_time(1);
        break;
    case 0x49:
        scsi_semaphore = 1;

        *(volatile unsigned char *)scsi_base = 0x10;
        *(volatile unsigned char *)scsi_reg = 0x30;

        while ( scsi_semaphore == 1 )
            eat_time(1);
        break;
    default:
        printf("Error: Unknown SCSI error\n");
        break;
    }
}

}
*(volatile unsigned char *)scsi_base = Ox12;
*(volatile unsigned char *)scsi_reg = Ox01;

*(volatile unsigned char *)scsi_base = Ox13;
*(volatile unsigned char *)scsi_reg = Ox00;

*(volatile unsigned char *)scsi_base = Ox14;
*(volatile unsigned char *)scsi_reg = Ox00;

*(volatile unsigned char *)scsi_base = Ox18;
*(volatile unsigned char *)scsi_reg = Ox09;

while ( scsi_semaphore == 1 )
    eat_time(1);
break;

default:
    printf("Unable to recover!\n");
    printf("\nStatus=%2X Message=%2X Phase=%2X\n",\n        scsi_num, scsi_message, scsi_phase);
    fflush(stdout);
    while (1)
        eat_time(1);
    break;
}

if (scsi_message == 0)
    command->cdb_status = SCSI_GOOD;
else
    command->cdb_status = SCSI_CHECK_SENSE;

return 1;
APPENDIX E
ADAPTER CARD SCHEMATICS

E.1 EV80960SX SCSI Adapter Card Schematics

The EV80960SX SCSI adapter card is fabricated using a standard four-layer printed circuit board technology. The inner layers are power and ground planes. The outer layers are signal planes. The prototype version had severe problems with ground bounce due to the length of the unshielded SCSI cable. To alleviate this problem, extensive numbers of decoupling capacitors were used. Figures E.1 through E.8 show the schematics of the SCSI adapter card.
E.2 EV80960SX LCD Adapter Schematics

The EV80960SX LCD adapter is also fabricated using a standard four-layer printed circuit board technology. To allow for easier debugging, all pins on the XC4005 are connected to test points for probing by a logic analyzer. Figures E.9 through E.14 show the schematics for the LCD adapter.
Figure E.10 Watchdog Timer
REFERENCES


