2003

EMRISC16 : an embedded risc microprocessor for low cost FPGA'S

James R. Petrus
Lehigh University

Petrus, James R.

EmRISC16: An Embedded RISC Microprocessor for Low Cost FPGA's

May 2003

EMRISC16: AN EMBEDDED RISC MICROPROCESSOR FOR LOW COST FPGA'S

by

James R. Petrus

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Engineering

Lehigh University

April 2003

This thesis is accepted in partial fulfillment of the requirements for the degree of Master of Science.

April 18, 2003
(Date)

Prof. Meghanad D. Wagh
(Thesis Advisor)

Prof. Donald M. Bolle
(Department Chair)

Acknowledgments

I would like to thank the following people regarding this thesis:

Meghanad Wagh, my thesis advisor and mentor, for being a fantastic instructor, a constant source of ideas, and encouraging me to struggle through many of the department's most challenging courses.

Terry Boult, my professor and research advisor, for employing me as a research assistant and teaching me how to deal with real-world engineering projects, where the problems are many and the goals unclear.

David Luksenberg, my housemate and fraternity brother, for enduring my play-by-play accounts of hardware debugging and offering practical solutions.

My other housemates and fraternity brothers, for their support over the years and the constantly reminding me that I do have to graduate someday.

James and Janet Petrus, my parents, for their unwavering support of my extended stay in academia.

Megan Petrus, my little sister, for advising me in many non-academic pursuits and reminding to me to take a break on the weekends.

Table of Contents

List of Tables

List of Figures

Abstract

The decreasing price and increasing power of FPGA's has made it practical to use an FPGA-based processor core in low production systems instead of using an existing microprocessor. Often, these systems emulate a popular processor and have application-specific peripherals implemented on-chip by the system designer. Most of these systems use powerful, 32-bit processors and are designed to do large amounts of DSP. But for low-end applications where processing power is not an issue, microcontrollers still dominate the industry. Additionally, many of these microcontrollers are derived from aging architectures, not designed with Internet communications in mind.

This thesis proposes a new, low-end microprocessor core with two major design goals. The first is that it has to be very simple. It should only support a minimal instruction set and be esily implemented on a low-end FPGA. The second goal is that despite the fact that a very minimal processor design is proposed, it must be powerful enough to communicate on the Internet.

Chapter 1

Introduction

This chapter serves as an introduction to embedded microprocessors. It first defines what is an embedded processor and then presents a justification for exploring low-end, embedded processor architectures. Next, some related work in the area of FPGA-based processors, systems-on-a-chip (SOC's), and Internet Protocol (IP) for microcontrollers will be presented. An outline of the remaining chapters of the thesis will conclude this chapter.

1.1 Overview of Embedded Processors

An embedded processor is a specialized microprocessor designed for use in a low cost system where a considerably more powerful PC or workstation processor would be impractical because of its cost, complexity, or power requirements. The realm

of embedded processors includes high-end application specific instruction processors (ASIP's) that are used in personal data devices, digital signal processors (DSP's) in cellular phones and other arithmetically intensive applications, and microcontrollers, which are most often used for controlling electro-mechanical systems or monitoring sensors.

It is with microcontrollers that this thesis is primarily concerned. Most are 8- or 16-bit machines that tend to run at clock frequencies less than 20 MHz. They are perfect for I/O intensive systems where speed is not a primary design goal. Although many of the most popular microcontroller architectures have remained largely unchanged for the last 20 years, they are still a relevant topic because of their wide variety if uses. Additionally, microcontroller sales made up 75% by volume of all microprocessors sold in 2002 [1].

In recent years, a major push has been made to make common appliances "smart" or "net-enabled." That is, allow them to communicate with each other or to a home PC, often using the Internet as the medium for that communication. The goal of this thesis is not to answer why one would want to put their toaster on the Internet, as this has already been done [2]. However, the ability of current microcontrollers to perform meaningful communication on the Internet is questionable.

1.2 Related Work

1.2.1 Microcontrollers and the Internet

There are several projects with the goal to give "TCP/IP functionality" to some of the more common 8-bit microcontrollers using the Crystal CS8900a Ethernet interface. The CS8900a is immensely popular for this type of application because it supports 8-bit data access mode and is available as a low cost, single chip solution.

A Norwegian consulting company, Systor Vest, manufactures CS8900a Ethernet modules and provides sample code for interfacing it with the Microchip PIC16C74 and the Atmel Atmega103 microcontroller [3]. Both example systems implement the Internet Control Message Protocol (ICMP) echo service, more commonly known as "ping." Although the Atmel microcontroller could easily handle a much larger portion of the IP stack, nothing more was implemented. On the other hand, their Microchip demonstration system could barely handle ICMP. It shares a problem common to many microcontrollers: it cannot address enough data memory to implement anything more complicated than network layer protocols such as IP and ICMP.

Using a CS8900a module similar to that manufactured by Systor Vest, two students at the University of California Riverside created a small system that uses an Intel 8051 microcontroller to communicate on the Internet [4]. Their system measures the room temperature and then sends it to another computer as a Unix

Datagram Protocol (UDP) packet. Although their UDP functionality is incomplete (send only) and they did not implement Address Resolution Protocol (ARP), their system is significant for two reasons. First, they are using the Intel 8051, which is often touted as "the world's most popular microcontroller," which also suffers from the low memory problem. The more important contribution is that they demonstrate a practical system where a very small, low power microcontroller needs to communicate on the Internet.

1.2.2 FPGA-based Processors

Relevant to the thesis as a whole but not to the idea of microcontroller-Internet communications is the growing popularity of using soft processor cores in embedded systems. A designer can purchase the rights to use a well-tested processor core, download the hardware description language (HDL) source code (usually VHDL or Verilog), and then customize the design to their specific needs. Often, a designer will add on-chip peripherals or perhaps some specialized DSP functional units. Once the design is done, the customized processor can be synthesized and fabricated. More typically, it is implemented on an FPGA to "future proof" the system – the processor itself can be changed or upgraded as new requirements for the system are introduced.

For high performance embedded systems, two companies that produce noteworthy soft cores are Tensilica Inc. (http://www.tensilica.com) and ARC International

(http://www.arc.com). Both the Tensilica Xtensa and the ARCtangent are 32-bit RISC processor cores designed for use in embedded systems. They are highly customizable; an embedded system designer can add multiply-accumulate units, choose among different integer multipliers, add single or double precision floating point units, and even choose the byte ordering scheme (big or little endian).

For low-end embedded systems where microcontrollers would be appropriate, the cores offered by Tensilica and ARC are overkill. The licensing fees are too expensive, not to mention the cost of the high gate count FPGA's that would be required. To make these low-end systems economically viable, the best choice for the designers is to use a free core from a website such as OpenCores (http://www.opencores.org). There are dozens of simple microcontroller cores and peripherals available, many of which can be implemented on inexpensive, low gate count FPGA's.

Particularly relevant to this thesis is the XSOC system-on-a-chip that uses the xr16 core developed by Jan Gray that was originally designed in a series of magazine articles [5], [6], [7]. The xr16 core is a 16-bit, pipelined RISC core targeted for the same development platform as the proposed processor to be introduced in the following chapter. Although a very well designed processor, the xr16 has a few flaws that make it less than ideal for use as a microcontroller. First, it was designed as part of a full SOC. It doesn't have peripheral I/O specific instructions, nor was it meant to interface with external chips. The second problem is that the XSOC can only address 64 KB of memory and I/O space. Considering that the onboard video

controller shares memory with the processor, this leaves on 32 KB of memory or I/O space available for programs and data. Finally, there is a licensing issue with the XSOC. It is restricted for use only for educational or non-commercial ventures [7], which prevents it from being widely used in production systems.

1.3 Organization of Thesis

This thesis explores a new microcontroller architecture that is powerful enough to communicate on the Internet, yet simple enough to be implemented on an inexpensive FPGA. Chapter 2 will examine the design goals and constraints of the processor, as well as area and delay figures for the FPGA on which it is implemented. Chapter 3 is a discussion of the processor's instruction set architecture (ISA), including details about its capabilities and internal architecture. Chapter 4 takes a look at how to interface the processor with an Ethernet module and other chips, and presents an example system to demonstrate its functionality. Chapter 5 will then conclude the thesis with a look at some limitations of the processor and suggests some future work to improve upon these limitations.

Chapter 2

Overview of the EmRISC16 Processor

Now the thesis will propose a new microcontroller design, the 16-bit Embedded RISC Processor (EmRISC16). Since subsequent chapters will focus on the implementation and functionality of the processor, this chapter is concerned with its design decisions. In particular, Chapter 2 will discuss the design goals for the EmRISC16, some constraints placed on the new design, and finally some details of its first FPGA implementation.

Table 2.1: EmRISC16 Features

18-bit Address Bus
16-bit Data Bus
16 General Purpose Registers
2 External Interrupts
von Neumann Architecture
47 Instructions: ALU, Branch, and I/O
No Multiplication nor Division Ops
No On-core Peripherals

2.1 Design Goals

As eluded to in Chapter 1, the overall goal of the EmRISC16 processor is twofold. First, it has to be powerful enough to perform significant communication on the Internet. This thesis defines significant Internet communication as not only the ability to interface with a high-speed network adapter (such as Ethernet), but also the ability to implement transport and session layer protocols such as TCP/IP or UDP/IP. The second goal is that the processor core has to be simple enough to implement on a low-cost FPGA, allowing room for it to be customized with additional on-chip peripherals.

A summary of the EmRISC16's capabilities is shown in Table 2.1. To support the goal of Internet communications over Ethernet, a lot of memory is needed handle incoming and outgoing packets. Using an 18-bit instead of 16-bit address bus allows for the much needed 256 K of address space. Additionally, external address and data buses are required to interface with an Ethernet adapter. A port-based, Harvard

architecture as used by many microcontrollers would be inadequate. Therefore, a von Neumann architecture with no internal memory was chosen.

Also consistent with the goal of creating a processor powerful enough for Internet communications is the choice to make EmRISC16, as the name implies, a 16-bit system. All the registers are 16 bits wide, as is the data bus. Often, an Internet application needs to keep track of numbers much larger than 256; Ethernet frames can be up to 1518 bytes in length. Various fields in the IP header require two bytes of storage such as the length, identifier, and checksum.

Finally, it was crucial that the EmRISC16 core not include any peripherals beyond what is necessary for a 16-bit RISC processor to adequately function. This maximizes the number of ways that someone could customize an implementation of it with whatever supporting circuitry is desired. Cascading interrupt controllers, parallel and serial ports, or other specialized hardware can be added with minimal difficulty.

2.2 Constraints

The goal of implementing EmRISC16 on a low-cost FPGA imposes many constraints on what features can be included as part of the processor. The most significant of these is the amount of logic that can fit on a small FPGA. It would be impossible to include multiplication or division hardware. Also, many of the processor's functional

units need to be as simple as possible - a tradeoff of execution speed for more FPGA logic blocks.

Besides the area constraints imposed by FPGA's, there is a set number of pins on an FPGA's package, many of which are dedicated to power, ground, or programming the chip. Furthermore, the first incarnation of EmRISC16 is implemented on a development board, which has its own pin restrictions. There are often memory chips, displays, and PC interface ports, all connected to the FPGA that dictate how the user accessible pins can be used.

A final constraint on FPGA implementation of the EmRISC16 processor is that it is limited by how fast the FPGA can be clocked. IC's implemented on FPGA's are going to be considerably slower than those fabricated on a silicon die. Also, just because a particular FPGA advertises that it can be clocked at a certain rate does not mean that a design implemented on it can run at that rate. This number refers to the speed that a single logic gate or flip-flop can operate without generating glitches. Said another way, a design with a 10 gate maximum delay can be clocked at 1/10 of the advertised rate.

2.3 Implementation

Because of it's availability and low cost, the EmRISC16 processor is implemented on an XS40-010XL+ Prototyping Board manufactured by the XESS Corporation

Table 2.2: Logic Capacity of the Xilinx XC4010XL FPGA

Max Logic Gates	10,000
Max RAM Bits	12,800
Typical Gate Range	7,000 - 20,000
Total Flip-Flops	1,120

(http://www.xess.com). It features a Xilinx XC4010XL FPGA, 128 K of SRAM, a programmable clock generator, a 7 segement display, an Intel 8031 microcontroller, PS/2 and VGA ports, and a parallel port for interfacing with a PC. Although the board has been discontinued by XESS, it used to sell for $209, making it one of the cheapest, albeit low-end, FPGA development systems available.

The XC4010XL is a relatively low-end, low-cost FPGA. It operates at 3.3 V and can be clocked no higher than 100MHz. It has a total of 400 Common Logic Blocks (CLB's) and 61 I/O Blocks (IOB's) [8]. A CLB on this FPGA consists of 2, 4-input lookup tables (LUT's) and 2 D flip-flops. An IOB contains an input buffer and D flip-flop and an output buffer and D flip-flop. Also available are 880 tristate buffers (TBUF's) and 8 global clock buffers (BUFGLS's). Table 2.2 shows an estimation of how much logic it can support [8]. The version of the chip used on the XESS board is an 84 pin package, and only 61 of these pins may be used in a design.

Synthesis and implementation results for the EmRISC16 on the XC4010XL are presented in Table 2.3. Synthesis was performed using Xilinx Foundation 2.1i, Student Edition. Note that although the processor could be clocked at almost 9.5 MHz,

Table 2.3: EmRISC16 Synthesis Results

Number of CLB's	399 out of 400
	86 Flip-flops
	637 4-input LUT's
	224 3-input LUT's
	32 Dual Port RAM's
Number of IOB's	48 out of 65
	14 Flip-flops
	0 Latches
Number of TBUF's	88 out of 880
Number of BUFGLS's	2 out of 8
Total Equivalent Gate Count	10,008
Maximum Frequency	9.453 MHz

the processor is usually run at exactly 9.0 MHz.

Chapter 3

The EmRISC16 Instruction Set Architecture

This chapter presents in detail the Instruction Set Architecture (ISA) of the EmRISC16 processor core. First, the capabilities of the processor are examined: what operations it supports, and on what types of data it can operate. Next, an in-depth look at each of the different functional units will be presented. The chapter will conclude with a brief overview of the EmRISC16 assembly language.

3.1 Operational Capabilities

The EmRISC16 ISA is many ways a little brother to the DLX Architecture [9]. It only supports 8- and 16-bit data types, is not pipelined, and has no floating point

unit. However, it has a similiar instruction set that implements most of the key operations needed for a modern RISC processor.

3.1.1 Registers

Sixteen 16-bit registers comprise the EmRISC16 register file: r0, r1, ..., r15; the value of r0 is always 0. Three other registers can also be indirectly modified by the programmer. They are the Program Counter (PC), the Interrupt Program Counter (INTPC), and Enable Interrupts Bit (EIB). Both the PC and INTPC are 18-bit registers that control program flow. PC always contains the address of the next byte to be fetched, while the INTPC will store the value of the PC when an external interrupt or trap occurs. The EIB can be set or cleared to enable or disable interrupts.

3.1.2 Data and Addressing

The EmRISC16 processor operates exclusively on 16-bit data words. However, when performing memory or I/O operations, it supports both 8-bit bytes and 16-bit words. This is done by providing instructions to sign-extend or zero-fill the high byte when reading bytes, and to write either the most signifcant or least significant byte of a register when performing a store. Also, note that 16-bit memory and I/O operations use Big Endian byte ordering.

The only data addressing modes supported by the processor is displacement.

That is, all memory and I/O operations refer to an address by providing an 18-bit base address and a register name that acts as a 16-bit offset to that address. Register deferred addressing can be achieved by providing 0 as the base address. Likewise, direct addressing is done by using r0 as the offset register. In addition to the displacement mode offered by the memory and I/O operations, all of the ALU operations support a destination register and two operand registers, commonly known as register addressing mode. Most of them also support the use of a 16-bit constant instead of a second operand, known as immediate addressing mode.

Program addressing is also very simple in the EmRISC16 architecture. All addresses are 18-bit immediates, removing the need to support PC-relative branches. Thus, a call to an address will refer to a full 18-bit address. A jump to an address stored in a register will result in the register's value shifted 2 bits the left being stored in the PC.

3.1.3 EmRISC16 Operations

The EmRISC16 ISA supports four types of operations: memory and I/O, ALU, branches and jumps, and system control. The memory and I/O operations are the loads and stores; currently implemented are instructions to read and write both bytes and 16-bit words to and from memory. Also, an I/O read and I/O write instruction were created to support slower, 8-bit periperhals. ALU operations consist

of the usual meddly of data manipulation operations: addition, subtraction, logical and arithmetic shifts, boolean operations, and numeric comparison operators. The branch and jummp instructions allow for subroutine calls, address and register jumps, and conditional brnaches. Finally, system control operations refer to the instructions that might be used by a real-time operating system (RTOS). These are the are the instructions that are used for managing interrupts and halting the processor. The full instruction set is provided in Appendix A.

3.2 Functional Units

As shown in Figure 3.1, the EmRISC16 processor consists of six functional units: the Fetch and Decode Unit, the Register File, the Arithmetic Logic Unit, the Program Counter, the Memory and I/O Unit, and the Processor Control. The first five units define the architecture and implement the various instructions. The sixth one is the control unit that is responsible for timing external signals and clocking registers when appropriate.

3.2.1 Fetch and Decode Unit

As its name suggests, the Fetch and Decode Unit (FDU) is responsible for fetching instructions from external memory and decoding them into signals usable by the rest of the system. It is shown in Figure 3.2. The FDU expects to be read data from an

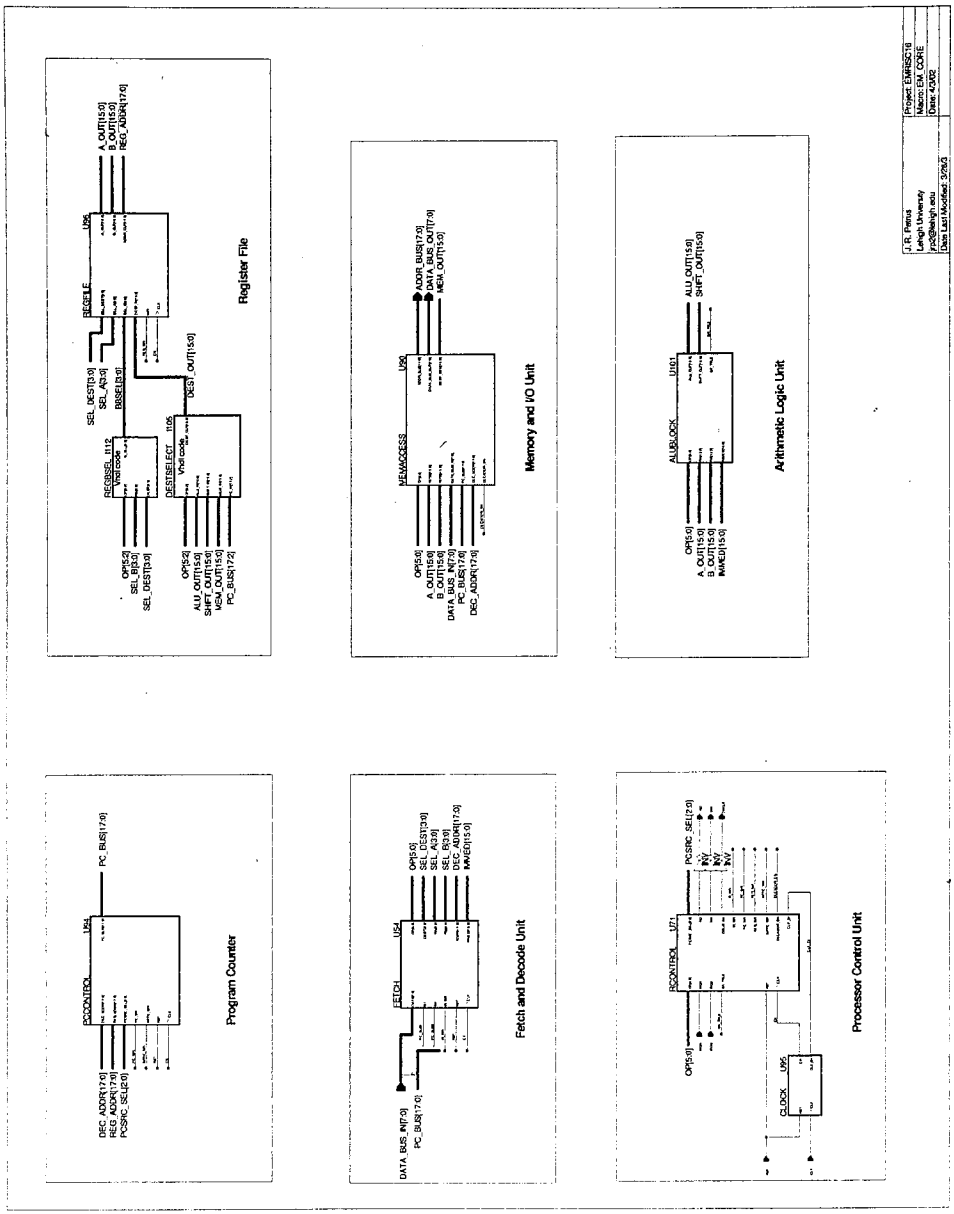

Figure 3.1: EmRISC16 Processor Core

8-bit external memory (either a ROM or an SRAM), and that all the instructions are 32-bit aligned.

The EmRISC16 processor uses a fixed instruction length. All instructions are exactly 32 bits in length and each operand (register, immediate, or address) has a fixed location in the instruction word. Although this does waste some space in memoruy, it makes the FDU an extremely simple unit and allows the encoding of 16-bit immediates and 18-bit addresses into the instruction word. Each of the instruction word formats is shown in Figure 3.3. Once the instruction word is fetched, the six VHDL blocks on the far right of the schematic re-wire the outputs of the four instruction registers into buses that carry the opcode and operands to the rest of the functional units.

3.2.2 Register File

Shown in Figure 3.4, the Register File Unit contains the 16 general purpose registers. Of these, register r0 is used to represent 0. In other words, anytime this register is read a value of zero will be retured. This is done so that a memory access is not necessary in order to set a register to 0 and is consistent with prevalent trends in RISC architectures. Thus r0 implementation does not require any flip-flops. It has a few peculularities in its design because it is specially designed for implementation on a Xilinx 4000 series FPGA. The most noticeable difference from a common register

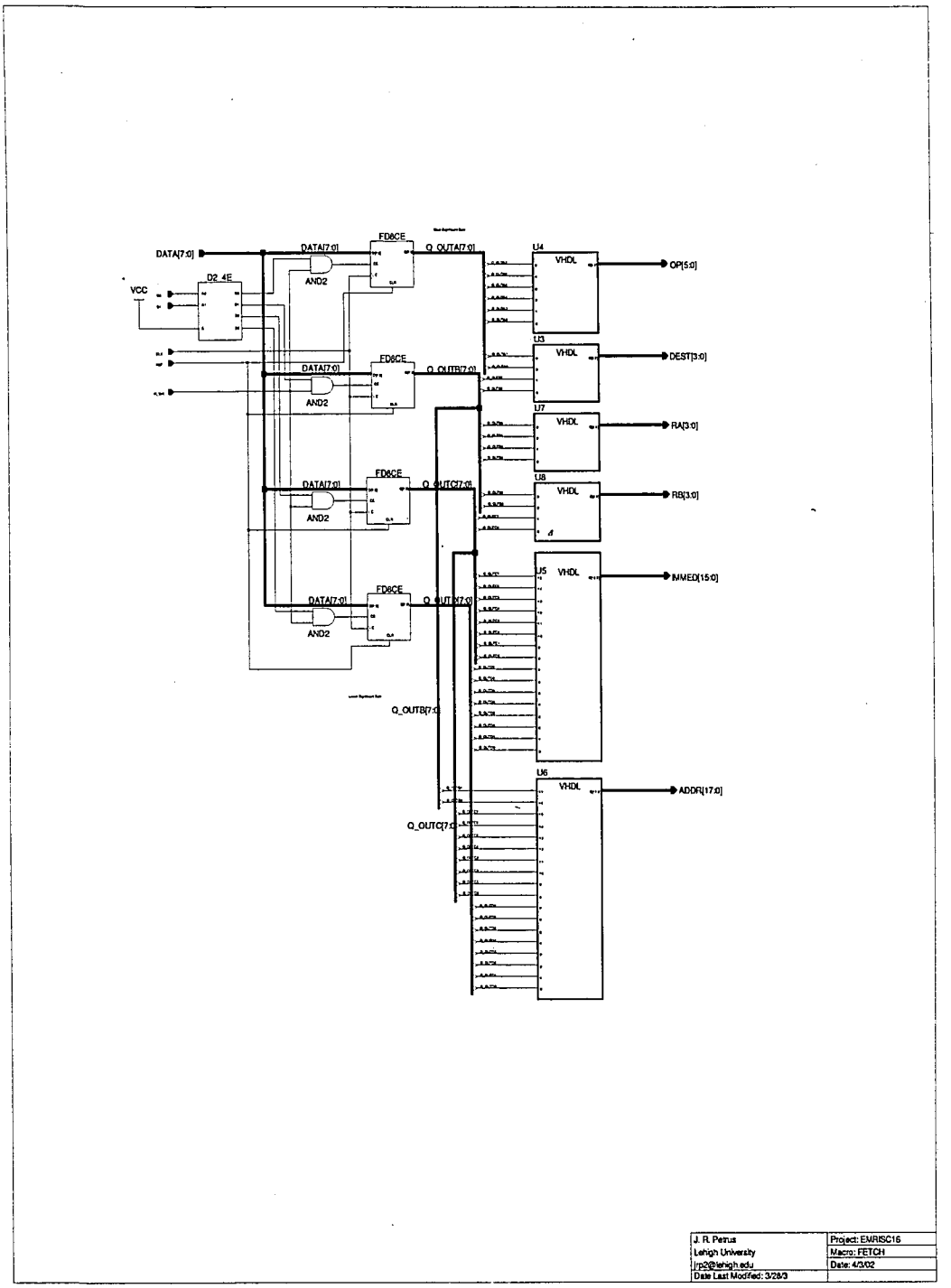

Figure 3.2: Fetch and Decode Unit

Opcode	Rd	Ra	Rb	Unused
6	4	4	4	14

a) ALU-Register Instruction Format

Opcode	Rd	Ra	Unused	Immed
6	4	4	2	16

b) ALU-Immediate Instruction Format

Opcode	Rd	Ra	Addr
6	4	4	18

c) Memory-I/O Instruction Format

Opcode	Unused	Addr
6	8	18

d) Address Jump Instruction Format

Opcode	Rd	Ra	Unused	Immed
6	4	4	4	16

e) Register Jump Instruction Format

Opcode	Unused	Ra	Address
6	4	4	18

f) Conditional Branch Instruction Format

Opcode	Rd	Unused	Address
6	4	4	18

g) Address Call Instruction Format

Opcode	Rd	Ra	Unused
6	4	4	18

h) Register Call Instruction Format

Opcode	Unused
6	26

i) Opcode Only Instruction Format

Figure 3.3: EmRISC16 Instruction Formats

file is that the register contents are stored in dual ported RAM's instead of D flip-flops. The reason for this is that 16x1 bit RAM elements map nicely to Xilinx LUT's, much more efficiently than the 240 flip-flops that would be required. The second difference is that this register file actually requires twice the storage that it should. It is configured so that two registers could be read and a third written all in the same clock cycle. To to this, the two source registers are each read from seperate pairs of 16x8 bit RAM elements. The result of the operation is then written back to both register sets simultaneously. The third oddity of the register file is that although a value could be written to r0, it cannot be read because a multiplexer will select a zero constant everytime r0 is requested. In practice, the assembler (see Appendix C) will prevent the programmer from attempting such an operation.

3.2.3 Arithmetic Logic Unit

The ALU provides the datapaths needed to implement arithmetic, boolean logic, shift, and conditional branch instructions. Figure 3.5 shows the three main subunits that comprise the ALU. Element U97 signals the processor control whether or not a branch shoudl be taken. It does this by making sure that the opcode is a branch instruction, and then checking the value of the register to see whether or not it is zero. Element U98 is written in VHDL and performs the various shift operations of which the EmRISC16 processor is capable. They are arithmetic and logical shifts, to either left or right, by any number of bit between 0 and 15, specified in either

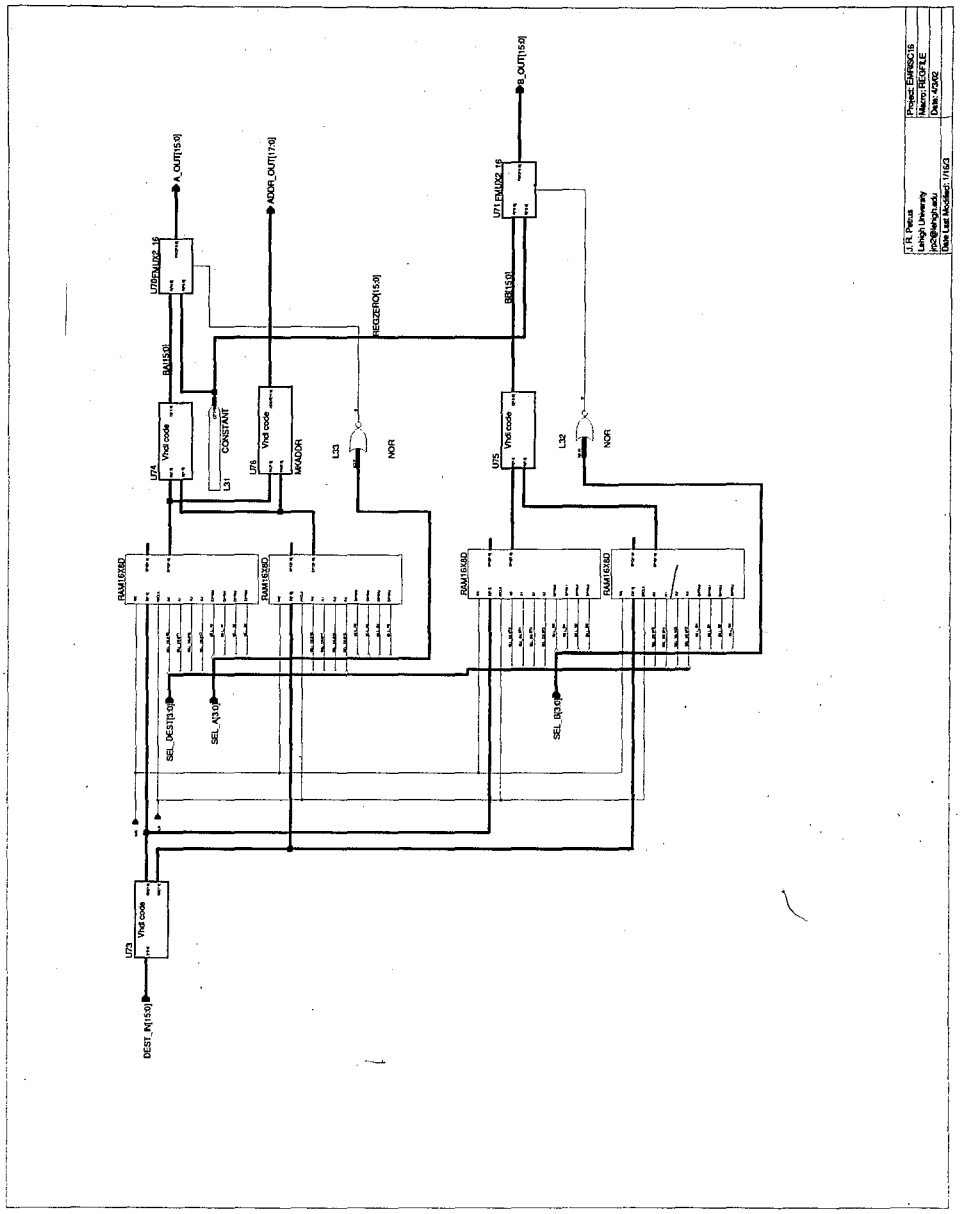

Figure 3.4: Register File

a register or an immeidate. The final subunit, U100, implements the arithmetic, boolean, and comparison instructions. It is shown in more detail in Figure 3.6.

3.2.4 Program Counter Unit

The Program Counter Unit, Figure 3.7, manages from what address the processor will fetch the next instruction. It mainly consists of the PC and INTPC registers, as well as a multiplexer to select what value should next be latched into the PC. This value can be any of six possibilities: PC+1, an address from the FDU, an address from the Register File, the value of INTPC, 0x10 (interrupt A), or 0x20 (interrupt B).

3.2.5 Memory and I/O Unit

Shown in Figure 3.8, the Memory and I/O Unit controls what address appears on the address bus, and manages bytes moving to and from the Register File or FDU. The adder (L31) finds the sum of a given register and an address from the FDU to implement displacement address mode. Multiplexer L32 determines which of two possible addresses appears on the address bus: an address from the PC for a fetch operation, or an address from L31 for a load or store operation. Finally, VHDL elements U88 and U89 perform the details of converting data types between bytes and 16-bit words. Note that in this implementation of the EmRISC16 processor, 16-bit loads and stores are not supported and only an 8-bit external data bus is

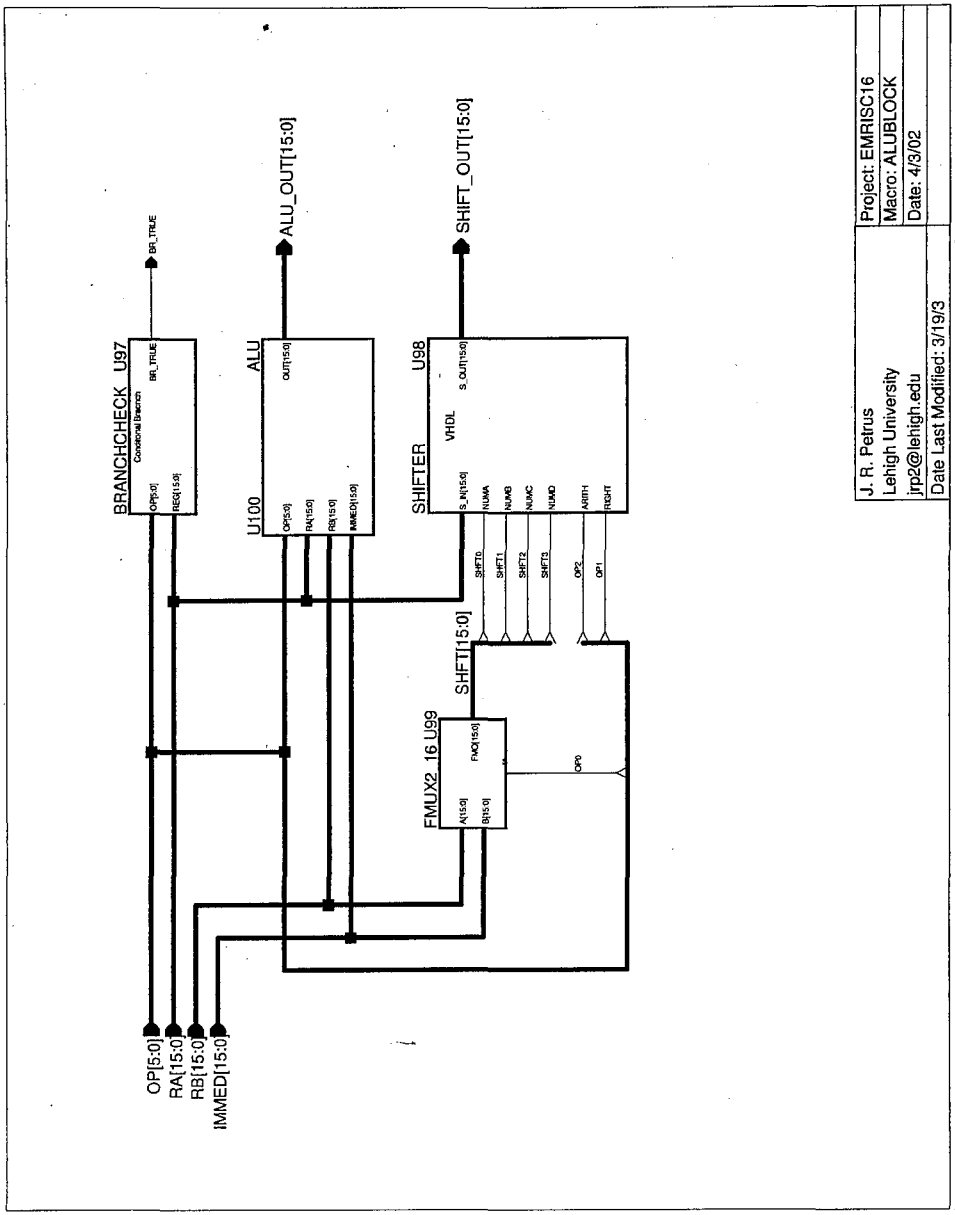

Figure 3.5: Arithmetic Logic Unit Block

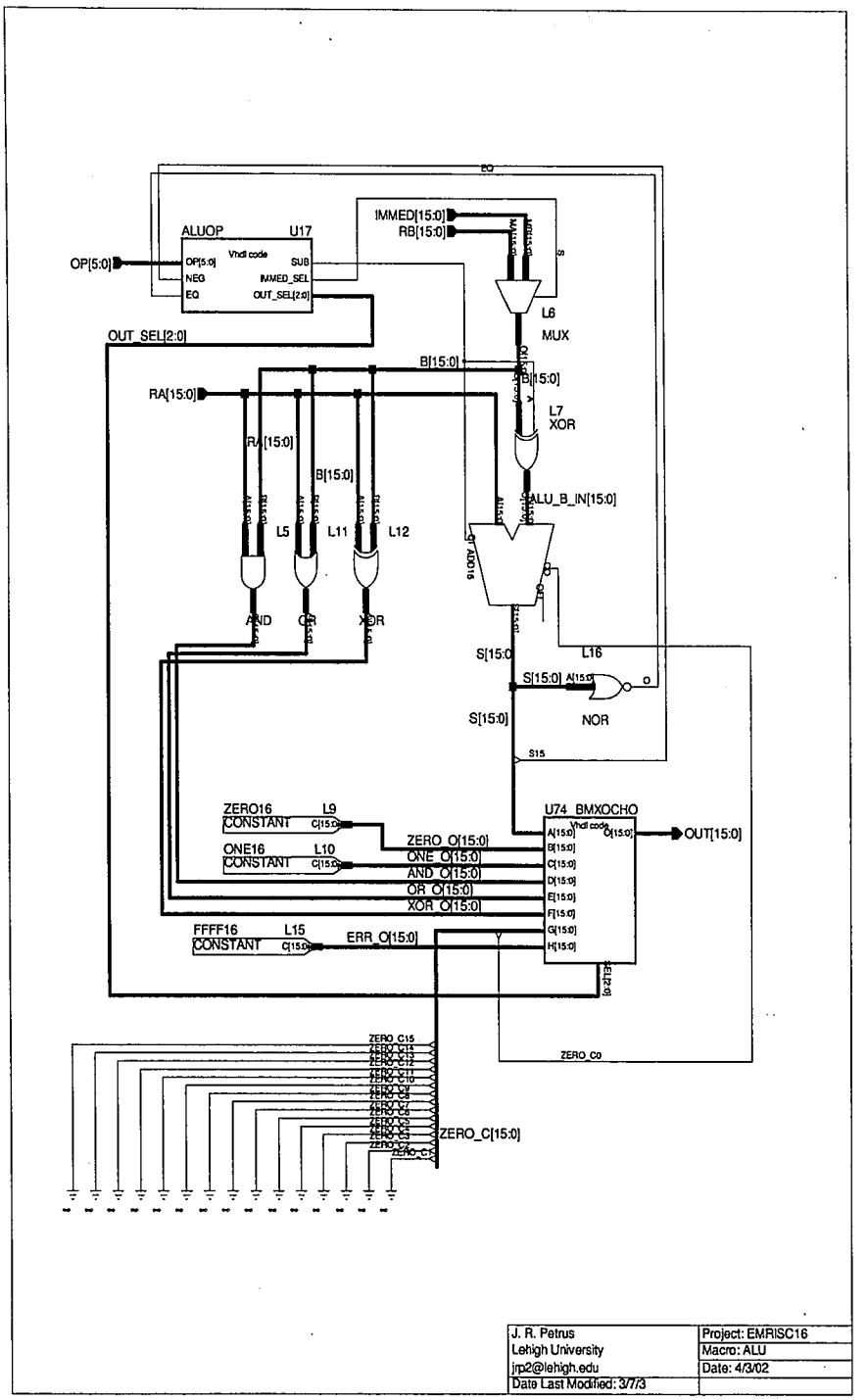

Figure 3.6: Arithmetic Logic Unit

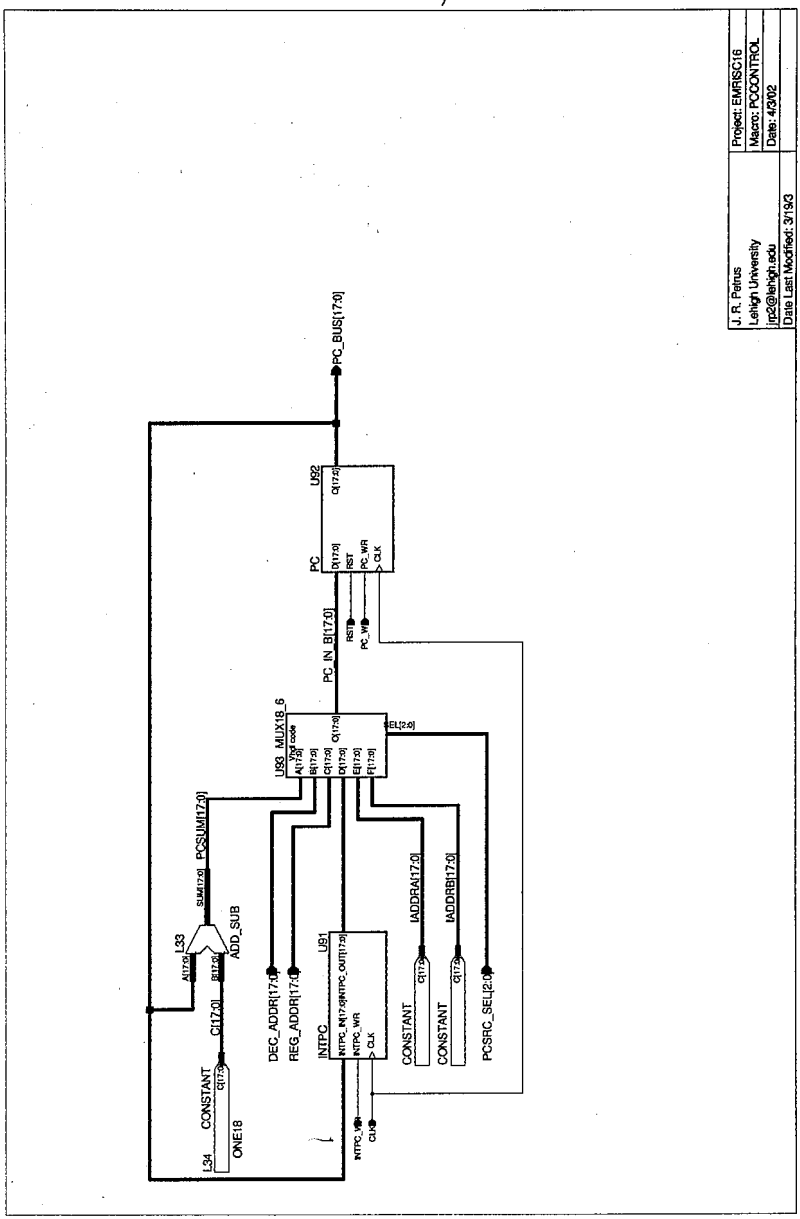

Figure 3.7: Program Counter Unit

present. This is because the demonstration system presented in Chapter 4 does not have any 16-bit peripherals. Additionally, this design choice makes 8 more pins available for other uses.

3.2.6 Processor Control Unit

The Processor Control Unit (Figure 3.9) consists of three flip-flops and VHDL code element U84. The upper two flip-flops are used as edge detectors to immediately latch incoming interrupts as soon as the inputs their clocks transition to high. The lower flip-flops is the EIB. Incoming interrupts are only processed if this bit is set. The VHDL code element is a state machine that controls the clocking of most of the processor's registers and outputs the read and write signals needed for peripheral communications.

3.3 EmRISC16 Assembly Language

No processor would be particularly useful without an assembler, so this chapter concludes with an overview of the EmRISC16 assembly language, which was developed as part of this thesis. As mentioned earlier, the EmRISC16 assembly language is very similiar to DLX and its derivatives (such as MIPS). Each instruction is written in all lowercase, and register names are r0, r1, ..., r15. A Table 3.1 contains

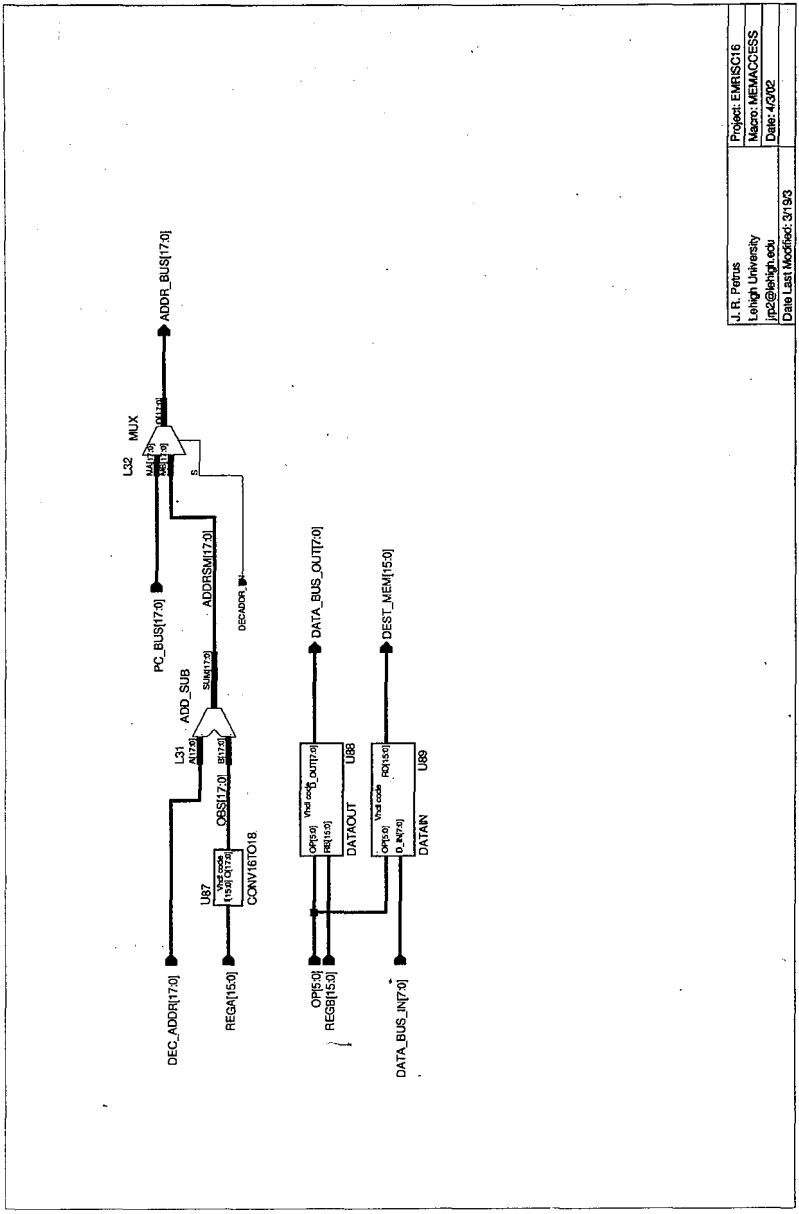

Figure 3.8: Memory and I/O Unit

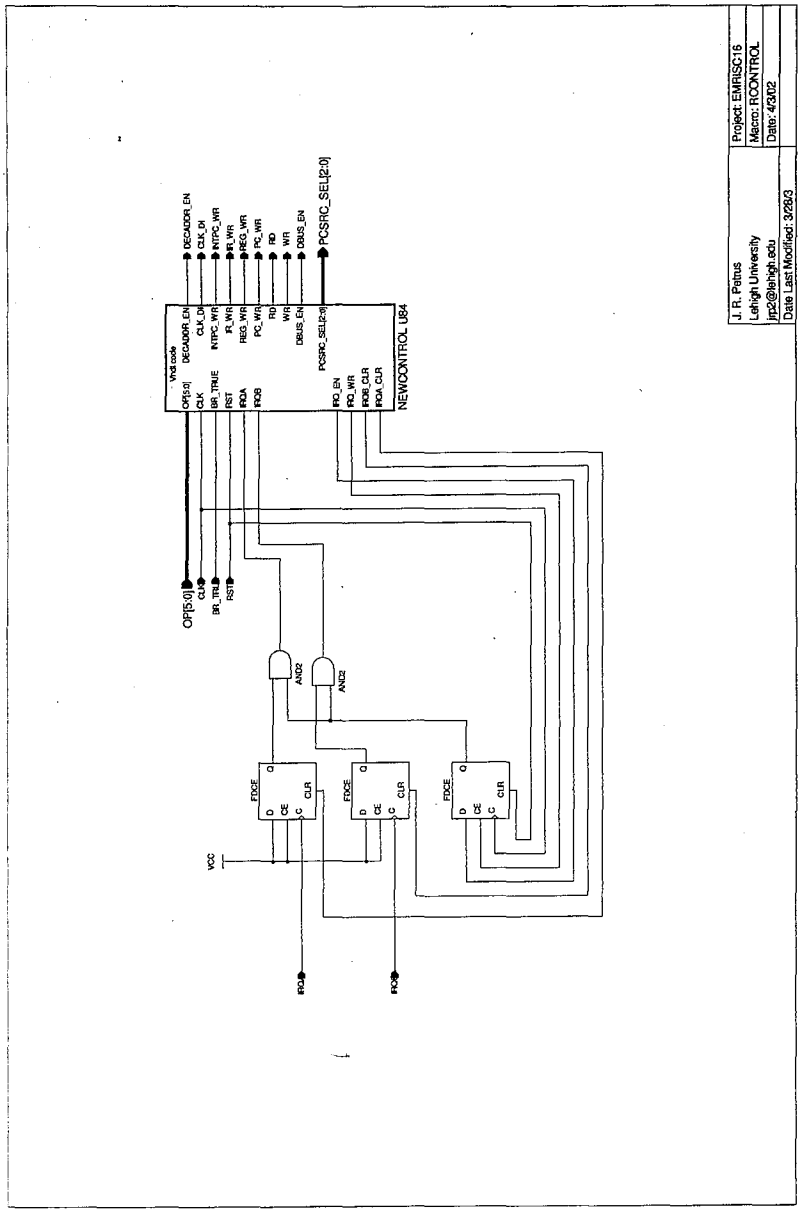

Figure 3.9: Program Control Unit

Table 3.1: Assembly Instruction Formats

Instruction Type	Example	Instruction Word Format
ALU-Register	add r2, r3, r4	Figure 3.3 a)
ALU-Immediate	subci r5, r5, r1	Figure 3.3 b)
Memory-I/O	sbh r3, 0x60(r1)	Figure 3.3 c)
Address Jump	ja 0x0	Figure 3.3 d)
Register Jump	jr r15 r5, r5, r1	Figure 3.3 e)
Conditional Branch	bnez r1, 0x10	Figure 3.3 f)
Address Call	acall r15, 0x30	Figure 3.3 g)
Register Call	rcall r14, r7	Figure 3.3 h)
Opcode Only	rfe	Figure 3.3 i)

a listing of the possible instruction formats. Both decimal and hexidecimal numbers are recognized; decimal numbers are written normally, but hexidecimal values must be prefixed with "0x." Labels must begin with an alphabetic character, but the remainder of its characters can be alphanumeric. The last character of a label must be a colon. The assembler also supports a number of directives for placing and defining data. These are listed in Table 3.2. Also, the full instruction set is in Appendix A. An example assembly program and corresponding Intel .HEX output are provided in Figure 3.3. Details about how to obtain the assembler and example programs are in Appendix C.

Table 3.2: Assembler Directives

Directive Usage	Description
.align n	Align next address with n bytes
.org n	Next address is n
.ds n	Skip n bytes
.db n	Define byte with value n at current address
.dw n	Define 16-bit word with value n at current address
.set X n	Add symbol X with value n to symbol table
.end	Signals the assembler to stop

```
        .set x 1                        :0400000010000000EC
                                        :0400040020000030A8
start:  di                              :0400080000000000F4
        ja prog                         :04000C0000000000F0
                                        :0400100000000000EC
        .org 0x30                       :0400140000000000E8
prog:                                   :0400180000000000E4
        acall r15, doadd                :04001C0000000000E0
        halt                            :0400200000000000DC
                                        :0400240000000000D8
doadd:  addi r1, r0, x                  :0400280000000000D4
        lbu r2, y(r0)                   :04002C0000000000D0
        add r3, r1, r2                  :040030003BC0003899
        jr r15                          :0400340004000000C4
                                        :0400380084400001FF
y:      .db 2                           :04003C0040800048B8
        .end                            :0400400080C48000F8
                                        :04004400243C000058
                                        :0400480002000000B2
                                        :00000001FF
```

Figure 3.10: An EmRISC16 Assembly Program and Resulting Intel .HEX Code

Chapter 4

Interfacing with the EmRISC16 Processor

Just as Chapter 3 discussed the ISA and internal architecture of the EmRISC16 processor core, this chapter will present the external features of the EmRISC16 processor. Section 4.1 looks at how to use the processor with other peripherals: the signals it generates and how they are timed. Then, Section 4.2 presents a full implementaion of a system that uses the EmRISC16 processor to communicate over the Internet.

Table 4.1: EmRISC16 Core Pinout

Pin Name	Purpose
CLK	Clock signal
RST	Reset signal
IRQA	Interrupt A
IRQB	Interrupt B
RD_	Read signal (active low)
WR_	Write signal (active low)
DBOUT_	Data output enable (active low)
ADDR_BUS[17:0]	Address bus
DATA_BUS_IN[7:0]	Incoming data bus
DATA_BUS_OUT[7:0]	Outgoing data bus

4.1 Memory and Peripheral Communications

As it was one the EmRISC16 processor's main design goals, it is an extremely simple to interface it with other chips. Table 4.1 shows the pinout of the processor's core. There are very few pins since it doesn't support special bus protocols (ISA, PCI, etc.), DMA, or DRAM. Quite simply, the processor communicates with other devices by putting the desired address on the address bus and activating either the RD_ or WR_ signals. The DBOUT_ signal is used to put data on the bus a few clock cycles before the peripheral is strobed by a falling WR_ pulse.

Perhaps the simplest peripheral communication that the EmRISC16 processor core performs is the fetch operation, shown in Figure 4.1. The fetch expects to communicate with an 8-bit wide memory device (SRAM or a ROM) and requires 8 clock cycles. The first, third, fifth, and seventh clock cycles assert the RD_ signal

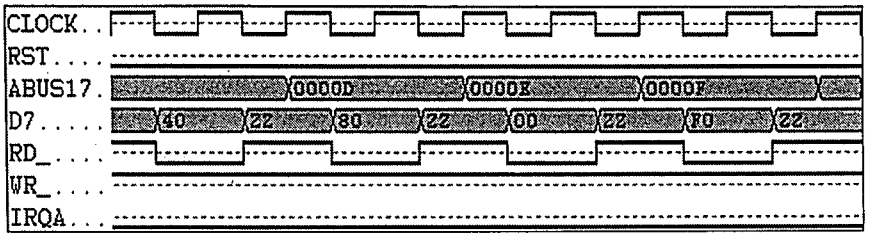

Figure 4.1: Instruction Fetch Timing

while internally the processor latches the data bytes on the positive clock edges. During the even numbered cycles, the PC register is incremented and the RD_ signal is deasserted. It should be noted that many SRAM chips, such as the one used on the Xess board, do not require the RD_ signal to be deasserted between subsequent address changes. Although use of this feature could reduce the fetch time to just 4 cycles, it may render the processor incompatible with other memory chips.

For communicating with fast external peripherals such as memory chips, the EmRISC16 processor uses the load and store instructions: lbu, lbs, lw, sbl, sbh, and sw. These instructions assume that the device can handle RD_ and WR_ signal assertions that are only one clock cycle in duration. The load instructions (Figure 4.2) all require 2 cycles beyond the 8 required for the instruction fetch. During the 9^{th} cycle, the desired address is put on the address bus. Then in the 10^{th} cycle, the address is still held on the bus, the RD_ signal is asserted, and the data byte is internally latched.

The store instructions (Figure 4.3) behave similiarly to the load instructions,

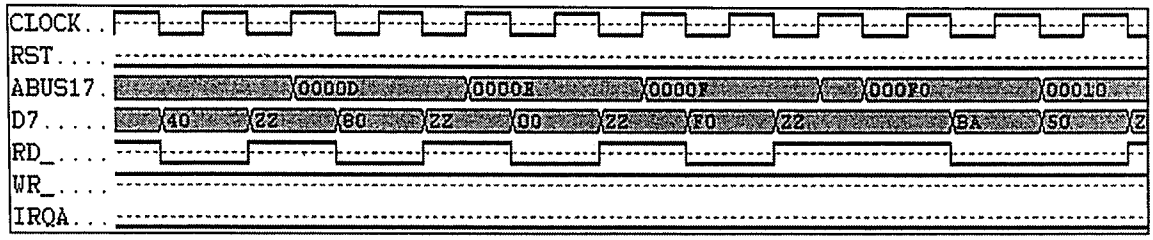

Figure 4.2: Memory Load Timing

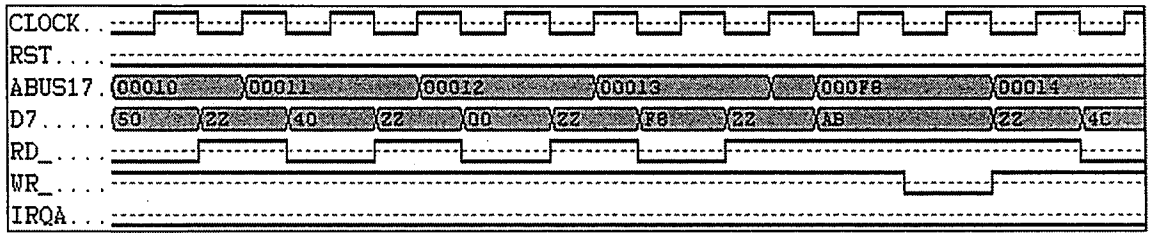

Figure 4.3: Memory Store Timing

but require 11 cycles in stead of 10. On the 9^{th} cycle the desired address is put on the address bus and the outgoing byte is placed on the data bus. During the 10^{th} cycle the WR_ signal is asserted and the address is still held. The 11^{th} clock cycle is needed so that there is a period of time between when the WR_ is deasserted and the RD_ is asserted for the fetch of the next instruction. Failure to insert this extra cycle results in the fetch operation incorrectly reading the byte that was just written.

Not originally part of the EmRISC16 ISA, the ior and iow instructions had to be added to allow communication with very slow peripheral devices. In particular, the CS8900a Ethernet adapter used in the demonstraton system requires RD_ and WR_ pulses that are well over 100 ns in length. So instead of slowing down the entire system to just 2 MHz, the ior and iow instructions were implemented.

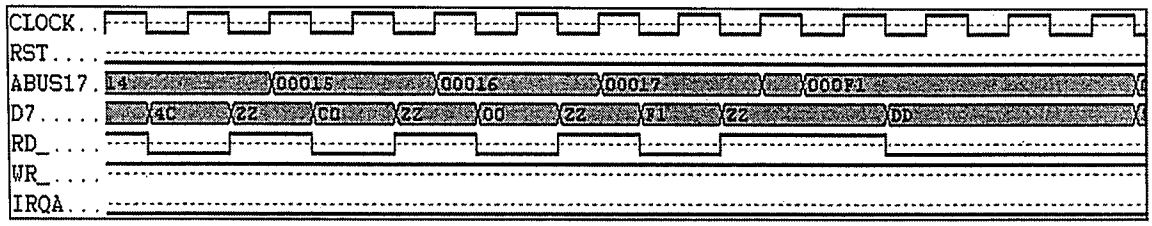

Figure 4.4: I/O Read Timing

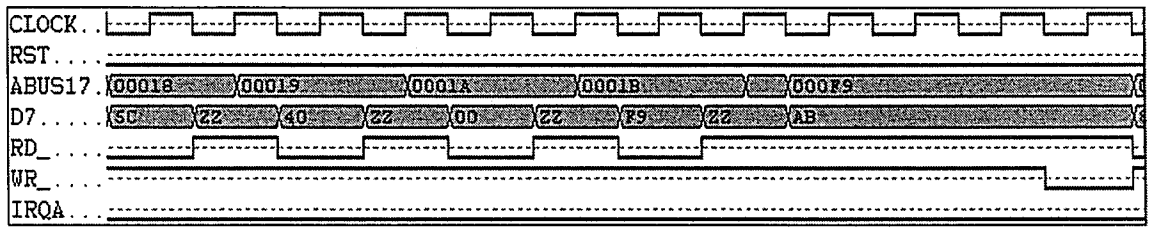

Figure 4.5: I/O Write Timing

The ior instruction, which reads a single byte from a peripheral, requires 12 clock cyles. Like the load instructions, the address is put on the bus during the 9^{th} cycle, but is held through the 12^{th} cycle. At the 10th cycle the RD_ signal is asserted and also held through the 12^{th} cycle. Data is actually latched by the processor during the 12^{th} clock cycle. Figure 4.4 shows the ior instruction timing.

The iow instruction, shown in Figure 4.5, also requires 12 clock cycles. The desired address and data are put on their respective buses on the 9^{th} through 12^{th} clock cycles. The WR_ signal is only asserted on the 12^{th} clock cycle. The extra clock cycle that was needed for the store instructions to deassert the WR_ is not needed for the iow instruction. This is because the subsequent instruction fetch would be from memory, not the slow peripheral to which a byte was just written.

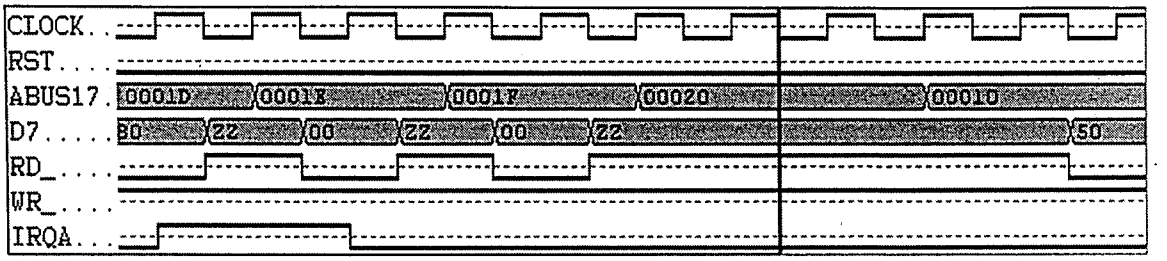

Figure 4.6: Interrupt Timing

Although not directly related to peripheral I/O operations, the timing of incoming interrupts deserve some mention. Incoming interrupts are always latched on the rising edge of the interrupt's pulse. However, they are only processed if the EIB is set and this processing occurs instead of performing an instruction fetch. Also note that interrupt A always had priority over interrupt B.

Once an incoming interrupt pulse is latched, the EmRISC16 processor requires three clock cycles to disable interrupts, save the current PC value in the INTPC register, and write the proper interrupt vector to the PC (0x10 for A, 0x20 for B). Figure 4.6 shows the processing of an interrrupt A. Although the interrupt occured during an addi instruction, it is not processed until the current instruction has finished execution. The vertical line in the figure indicates where the processor starts handling the interrupt. After the 3 cycles for the interrupt, the processor resumes fetching instructions at the new address.

4.2 Demonstration System

In the previous chapters, this thesis has proposed and discussed the EmRISC16 processor core with two main design goals. One, that it is powerful enough to perform meaningful communication on the Internet. And two, that it is simple enough to be implemented on an inexpensive FPGA. To prove that these goals were in fact achieved, this section presents a demonstration system that uses the EmRISC16 core to build a microcontroller that communicates using UDP/IP over Ethernet.

4.2.1 Microcontroller Functionality

The first step to making the EmRISC16 processor core usable is to add some additional functionality to make it a microcontroller. By only defining the EmRISC16 core, any incarnation of the processor can have an endless number of different on-chip peripherals. In the case of the demonstration system, two 8-bit ports (one input and one output) were added, as well as some additional address decoding logic to control the I/O ports and the external peripherals. Table 4.2 shows the additional pins on the EmRISC16 processor, as it appears when implemented on the FPGA. The UCRST is needed to hold the unused 8031 microcontroller that is on the Xess XS-40 board in reset. Also note that the 8-bit output port was renamed to LED[7:0] because it is used to drive to a 7-segment display. Figure 4.7 shows the top-level schematic of what is implemented on the FPGA.

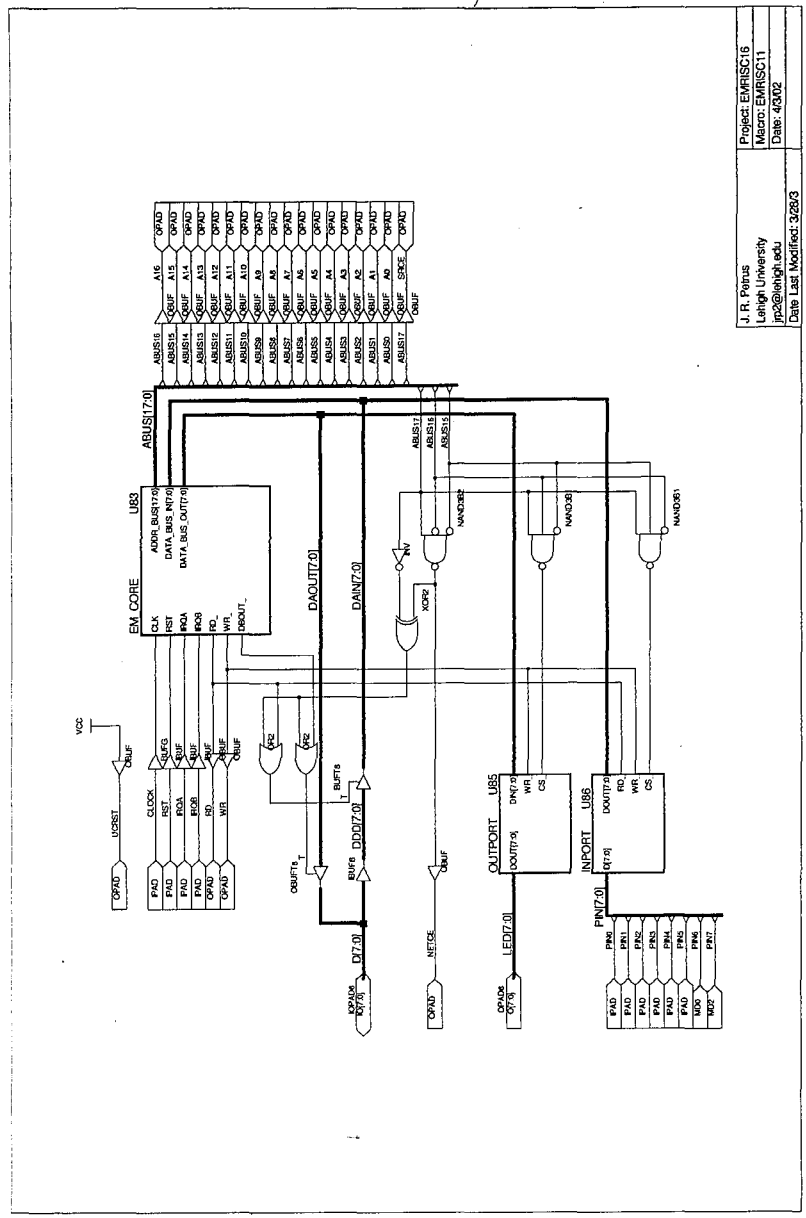

Figure 4.7: EMRISC16 Microcontroller

Table 4.2: EmRISC16 Microcontroller Pinout

Pin Name	Purpose
CLOCK	Clock signal
RST	Reset signal
IRQA	Interrupt A
IRQB	Interrupt B
RD_	Read signal (active low)
WR_	Write signal (active low)
A[16:0]	Address bus
D[7:0]	Data bus
SRCE_	SRAM chip enable (active low)
NETCE_	Network chip enable (active low)
PIN[7:0]	8-bit input port
LED[7:0]	8-bit output port (also 7-segment)
UCRST	8031 Reset

4.2.2 External Peripherals

Once the EmRISC16 processor core has been implemented as a microcontroller, some off-chip peripherals were needed to complete the demonstration system. Already present on the XS-40 prototyping board is an Alliance Semiconductor AS7C31024, which is a 128 KB SRAM. Also present on the board is the 7-segement display (connected to the 8-bit output port) and a parallel port connector that can be connected to a personal computer. The least significant 3 bits of the parallel port are connected to the EmRISC16 RESET, IRQA, and IRQB. The other 5 bits are connected to the correspoding bits of the 8-bit input port. The least significant 3 bits of the input port are tied to ground. Finally, an Ethernet module had to be provided for Internet communications. A Cirrus Logic CS8900a Ethernet module from Systor Vest. It

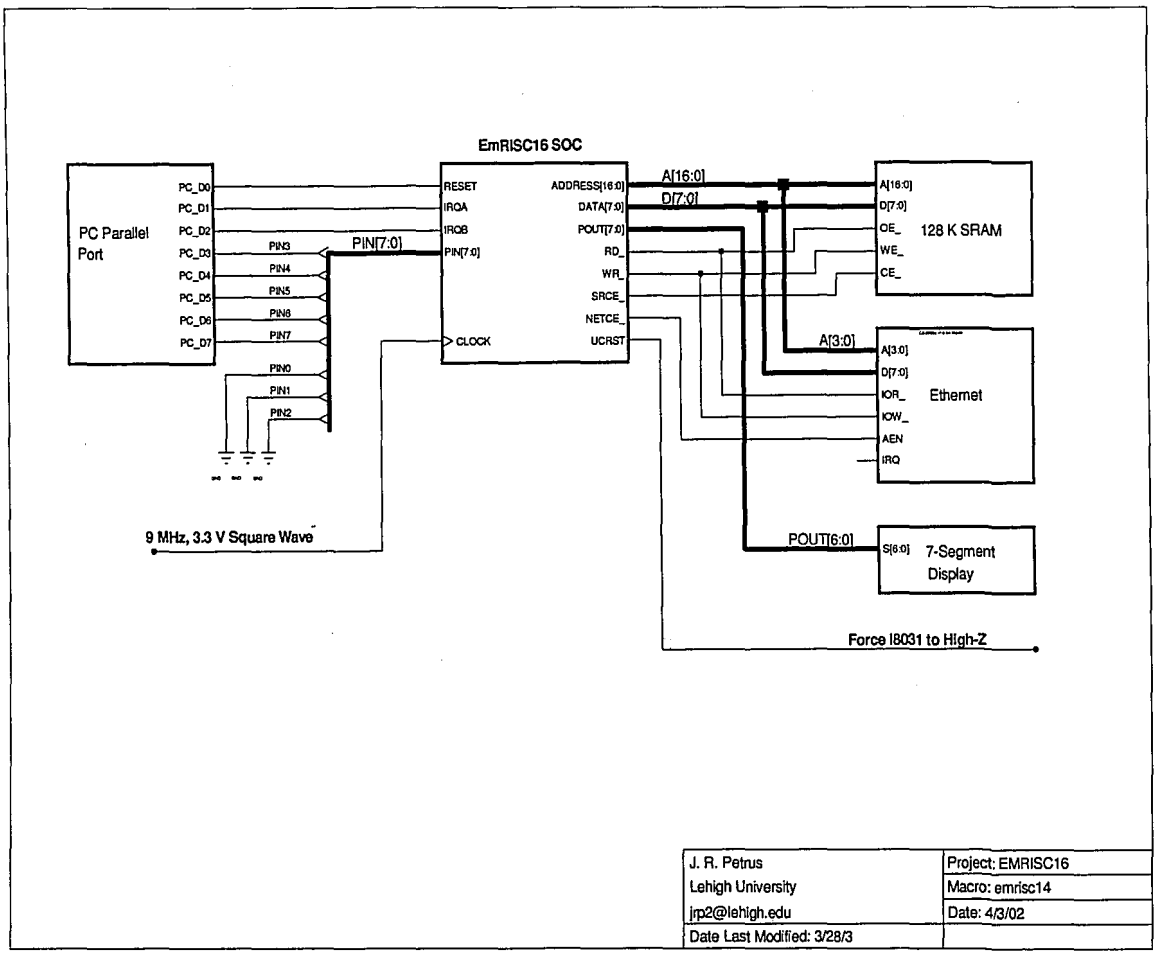

Figure 4.8: Demonstration System

provides an 8-bit wide data bus interface and communicates on 10BASE-T Ethernet at 10 Mbps. Figure 4.8 shows the relevant parts of the demonstration system and Table 4.3 shows the corresponding memory map.

4.2.3 Demonstration Program

With all the hardware in place, all the remains for discussion is the demonstration program. Written entirely in assembly, it uses just less than 5 KB for text and data

Table 4.3: Demonstration System Memory Map

Address Range	Peripheral
0x00000 - 0x1FFFF	128 KB SRAM
0x20000 - 0x2000E	CS8900a Ethernet
0x28000	8-bit Input Port
0x30000	8-bit Output Port

memory. The majority of the code consists of drivers to provide layers 1 - 4 of the OSI Network Architecture: Physical, Data link, Network, and Transport [10] and [11]. The Physical layer is, of course, encapsulated entirely in the Ethernet Adapter. The Data link layer consists of routines to send and receive Ethernet frames. The Network layer provides two key functions: ARP, used to resolve IP addresses to Ethernet MAC addresses, and partial IP functionality. IP packets can be sent and received, but fragmentation is not supported. Therefore, all IP packets must be smaller that 1500 bytes, which is the maximum frame size of Ethernet. Finally, the Transport layer adds the ability to demultiplex packets according to a port number. This is provided by implementing the UDP protocol.

The main routine of the demonstration program uses the network drivers to perform some trivial tasks, mainly to show that all the EmRISC16 processor's features are working. When the system comes out of reset, the first thing the program does is initialize the CS8900a and display a '0' on the 7-segment to indicate that everything is running. Once this has happened, it sends an ARP request to resolve IP address 192.168.0.1, the address of the computer with which it expected to communicate.

The program then jumps into an infinite loop of handling incoming frames and interrupts. An interrupt A causes the system to send a UDP packet stating "interrupt A" and then disable the handling of future interrupts on A; this is necessary to that it doesn't flood the network with UDP packets. An interrupt B re-enables interrupt A. Incoming ARP requests for the systems IP address, 192.168.0.2 are handled, and the first byte of incoming UDP packets on port 5000 is summed with the value on the input port, then shown on the 7-segment display. The full assembly source is included in Appendix B.

Chapter 5

Conclusion

Since all the work that has been done to create the EmRISC16 processor has been presented, the thesis will conclude with a few closing thoughts. First, the limitations of the processor must be considered. Then with these problems identified, examine how future versions of the processsor could be improved. Also, what other work could be done in the realm of net-enabled, FPGA-based microprocessors?

5.1 Considerations

Although the EmRISC16 processor core achieves its primary goal of being simple to implement, yet powerful enough to communicate on the Internet, it still leaves much to be desired in the areas of performance and of capabilites. The processor's performance suffers mainly because of its external data bus is currently only 8 bits

wide instead of 16. Granted, this choice was made because of the availability of 8-bit devices. The XS-40 prototyping board uses an 8- instead of a 16-bit SRAM. In addition, 16-bit versions of the CS8900a Ethernet module were not readily available for purchase. A custom PCB board would have to be built.

The EmRISC16 core provides most of the functionality expected from a simple processor core. However, its incarnation as a microcontroller on an FPGA leaves much to be desired. The two one-way 8-bit ports are acceptable, but two bi-directional 8-bit ports would be much better. In addition to I/O ports, the microcontroller needs one or two programmable interrupt timers. This is necessary because many higher level communications protocols (such as TCP) require time-out periods that cannot be easily implemented in software. Finally, a processor designed for Internet communications needs to support some type of encryption. Since most modern encryption algorithms require far more processing power than the EmRISC16 can provide, an on-chip encrytpion peripheral is needed to provide data security.

5.2 Future Research

In its current state, the EmRISC16 processor is well-suited for controlling a small emmbedded system that receives orders and sends status requests over the Internet using UDP/IP. It is also quite customizeable for a specific application because

only the processor core is strictly defined; additional peripherals could be added as needed, and a larger capacity FPGA can always be used. However, once a TCP/IP stack is written for it the EmRISC16 can be used for a much wider variety of applications.

A predicatable, but not very interesting application, would be to use the EmRISC16 in an embedded system that offered a web-based interface. Many embedded systems already do this. A ready example are the broadband firewall/router boxes that are currently available. Keeping in mind that the EmRISC16 is designed to be implemented on an FPGA, why not explore the idea of field-upgradeable embedded systems? Imagine using the EmRISC16 as the core of a small embedded system that communicates on the Internet, but is in a location where it would be difficult to access it for service. Assuming the hardware itself is intact, the system's Internet interface could be used to download new software and hardware revisions from a centralized server. Once the new versions have been downloaded, the EmRISC16 core re-programs its EEPROM's that contained the current software and hardware bitstream[1]. The processor triggers a sytem-wide reset and comes back online with all new software and hardware.

Also of great importantance to the EmRISC16's usefulness in embedded applications is the amount of power it consumes. Because it is implemented on an FPGA, it consumes much more power than it would if implemented in silicon. Regardless,

[1]A bitstream is a binary file that tells an FPGA how to configure itself.

a study of the EmRISC16's power consumption characteristics could show which instructions consume the most, and perhaps suggest ways to improve it without sacrificing too much performance.

Bibliography

[1] J. G. Ganssle, "Micro Minis," *Embedded.com*, March 2003.

[2] R. Southgate, "Toasty: A web enabled weather forecasting toaster." http://www.dancing-man.com/robin/webhome/report2.htm, 2001.

[3] G. T. Desrosiers, "Interfacing to the ATmega and 8515 Microcontrollers." http://www.embeddedethernet.com/appnotes/EmbEthAtmelSample.html, 1999.

[4] A. Gupta and P. Hoang, "An Internet Interface for Embedded Systems." http://www.cs.ucr.edu/ vahid/sproj/udpip/, 2001.

[5] J. Gray, "Building a RISC System in an FPGA. Part 1: Tools, Instruction Set, and Datapath," *Circuit Cellar*, vol. 116, March 2000.

[6] J. Gray, "Building a RISC System in an FPGA. Part 2: Pipeline and Control Unit Design," *Circuit Cellar*, vol. 117, April 2000.

[7] J. Gray, "Building a RISC System in an FPGA. Part 3: System-on-a-Chip Design," *Circuit Cellar*, vol. 118, May 2000.

[8] Xilinx, Inc., "XC4000E and XC4000X Series Field Programmable Gate Arrays." http://www.xilinx.com/bvdocs/publications/4000.pdf, May 1999.

[9] J. Hennesey and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition.* San Francisco, CA: Morgan-Kauffman, 1996.

[10] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approach, Second Edition.* San Francisco, CA: Morgan-Kauffman, 2000.

[11] J. Bentham, *TCP/IP Lean: Web Servers for Embedded Systems.* Lawrence, KS: CMP Books, 2000.

Appendix A

Instruction Set

The full EmRISC16 instruction set is provided in Tables A.1 - A.4. Note that *rd* is the destination register, *ra* is the first source register, and *rb* is the second. *Addr* refers to an 18-bit address and *Immed* is a 16-bit constant.

Table A.1: EmRISC16 Instruction Set: System Control Instructions

Instruction	Opcode	Clock Cycles	Description
nop	0	9	no op
halt	1	9	halt
rei	3	9	return from exception and enable interrupts
di	4	9	disable interrupts
ei	5	9	enable interrupts
trap	6	10	software interrupt
rfe	7	9	return from exception

Table A.2: EmRISC16 Instruction Set: Branch and Jump Instructions

Instruction	Opcode	Clock Cycles	Description
ja *Addr*	8	9	jump to *Addr*
jr *ra*	9	9	jump to address in *ra*
acall *rd*, *Addr*	E	10	call subroutine *Addr* and store PC in *rd*
rcall *rd*, *ra*	F	10	call subroutine in *ra* and store PC in *rd*
beqz *ra*, *Addr*	18	9	branch equal to zero
bnez *ra*, *Addr*	19	9	branch not equal to zero

Table A.3: EmRISC16 Instruction Set: Memory and I/O Instructions

Instruction	Opcode	Clock Cycles	Description
lbu *rd*, *Addr*(*ra*)	10	10	load byte unsigned
lbs *rd*, *Addr*(*ra*)	11	10	load byte signed
lw *rd*, *Addr*(*ra*)	12	10	load 16-bit word
ior *rd*, *Addr*(*ra*)	13	12	I/O read byte
sbl *rd*, *Addr*(*ra*)	14	11	store byte low
sbh *rd*, *Addr*(*ra*)	15	11	store byte high
sw *rd*, *Addr*(*ra*)	16	11	store 16-bit word
iow *rd*, *Addr*(*ra*)	17	12	I/O write byte

Table A.4: EmRISC16 Instruction Set: ALU Instructions

Instruction	Opcode	Clock Cycles	Description
add *rd, ra, rb*	20	9	add
addi *rd, ra, Immed*	21	9	add immediate
addc *rd, ra, rb*	22	9	add carry result
addci *rd, ra, Immed*	23	9	add carry result immediate
sub *rd, ra, rb*	24	9	subtract
subi *rd, ra, Immed*	25	9	subtract immediate
subc *rd, ra, rb*	26	9	subtract carry result
subci *rd, ra, Immed*	27	9	subtract carry result immediate
and *rd, ra, rb*	28	9	AND
andi *rd, ra, Immed*	29	9	AND immediate
or *rd, ra, rb*	2A	9	OR
ori *rd, ra, Immed*	2B	9	OR immediate
xor *rd, ra, rb*	2C	9	XOR
xori *rd, ra, Immed*	2D	9	XOR immediate
lsl *rd, ra, rb*	30	9	logical shift left
lsli *rd, ra, Immed*	31	9	logical shift left immediate
lsr *rd, ra, rb*	32	9	logical shift right
lsri *rd, ra, Immed*	33	9	logical shift right immediate
asr *rd, ra, rb*	36	9	arithmetic shift right
asri *rd, ra, Immed*	37	9	arithmetic shift right immediate
slt *rd, ra, rb*	38	9	set less than
sle *rd, ra, rb*	39	9	set less than or equal
sgt *rd, ra, rb*	3A	9	set greater than
sge *rd, ra, rb*	3B	9	set greater than or equal
seq *rd, ra, rb*	3C	9	set equal
sne *rd, ra, rb*	3D	9	set not equal

Appendix B

Demonstration Program

```
;; EmRisc16 UDP Demonstration
;; By J. R. Petrus, jrp2@lehigh.edu
;; 3/3/2003

;; conventions for ease of programming
;; r0 is always zero (defined by architecture)
;; r15 is function return address

;; 8 bit ports
.set    POUT        0x30000
.set    PIN         0x28000

;; Ethernet interface
.set        NET         0x20000
.set        RxTx0       0x20000
.set        RxTx1       0x20002
.set        TxCmd       0x20004
.set        TxLen       0x20006
.set        IStat       0x20008
.set        PPPtr       0x2000A
.set        PPD0        0x2000C
.set        PPD1        0x2000E

;; Internet Proto Type constants
.set        ARPN        0x0806
.set        IPN         0x0800
.set        UDPN        0x11
.set        IPVN        0x45        ; IPV4, hdr len = 5

;; Offsets that are related to ARP
;; relative to start of ethernet data
.set        HTYPE       0       ; hardware type
.set        PTYPE       2       ; protocol type
.set        HWADL       4       ; hardware address length
.set        PADL        5       ; IP address length
```

```
        .set        ARPOP       6       ; ARP operation
        .set        SHWA        8       ; sender HW address
        .set        SIPA        14      ; sender IP address
        .set        THWA        18      ; target HW address
        .set        TIPA        24      ; target IP address
        .set        ARPLEN      42      ; ARP packet length

        ;; Offsets that are related to IP
        ;; relative to start of Ethernet data
        .set        VHDRL       0       ; version and header length
        .set        SERV        1       ; service precedence
        .set        IPLEN       2       ; total datagram length
        .set        IDENT       4       ; Identification number
        .set        FLAGS       6       ; IP flags
        .set        TTL         8       ; time to live
        .set        PCOL        9       ; IP protocol, UDP=17
        .set        IPCSUM      10      ; checksum
        .set        IPSRC       12      ; Sender's IP address
        .set        IPDST       16      ; Receiver's IP address
        .set        IPDATA      20      ; the data, assuming no options

        ;; Offsets that are related to UDP
        ;; relative to start of Ethernet (not IP) data
        .set        SRCPORT     20      ; sender's UDP port
        .set        DSTPORT     22      ; receiver's UDP port
        .set        UDPLEN      24      ; length of UDP packet
        .set        UDPCSUM     26      ; UDP checksum
        .set        UDPDATA     28      ; UDP data

        ;; Port numbers that we shall use
        .set        UDPRCV      5000    ; listen for UDP on port 5000
        .set        UDPSND      5001    ; will send to UDP port 5001

        ;; disable interrupts for the moment (although the hardware
        ;; disables them by default on reset)
        ;; jump to the "main" program
start:  di
        ja prog

        ;; allocate space for IntA and IntB
        ;; remember that interrupts have space for exactly 4
        ;; instructions = 16 bytes each
        .org 0x10
intA:
        sbl r0, flagA(r0)
        rei

        .org        0x20
intB:
        sbl r0, flagB(r0)
        rei

        .org        0x30
prog:
        ;; light up the display with a 0
        add r2, r0, r0
        acall r15, led

        ;; initialize the 'net adapter
        acall r15, NetInit
```

```
        ;; we'll need to know the IP address of my laptop
        ;; so perform an ARP request
        acall r15, ARPReq

        ;; now we can safely enable interrupts
        addi r1, r0, 0xFF
        sbl r1, flagA(r0)
        sbl r1, flagB(r0)
        ei

        ;; forever loop, processing frames and interrupts
eLoop:
        ;; check to see if interrupt A has occured
        lbu r1, flagA(r0)
        bnez r1, checkB
        acall r15, intAf

checkB:
        ;; and for an interrupt B
        lbu r1, flagB(r0)
        bnez r1, checkNet
        acall r15, intBf

checkNet:
        acall r15, NetRecv

        ;; check the size in r1, if zero, try again
        beqz r1, eLoop

        ;; r1 != 0, therefore we have a frame
        ;; 1st see if ARP packet
        addi r3, r0, ARPN
        seq r4, r2, r3
        beqz r4, IPHandle

        ;; it is ARP, let's handle it
        acall r15, ARPFunc
        ja eLoop

IPHandle:
        addi r3, r0, IPN
        seq r4, r2, r3
        beqz r4, eLoop

        ;; it's an IP packet
        acall r15, IPFunc
        ja eLoop

led:
        ;; this subroutine looks at a value in r2
        ;; and then writes what should be output
        ;; to a 7-segment display to POUT
        andi r2, r2, 0xF
        lbu r3, ledda(r2)
        sbl r3, POUT(r0)        ; light it up
        jr r15

        ;; interrupt A handler
        ;; when we get an intA, send a UDP packet to the laptop
```

```
intAf:
        ;; 1st, make sure interrupt A is enabled
        lbu r1, enableA(r0)
        bnez r1, goA
        jr r15

goA:
        ;; now disable interrupt A until B occurs to
        ;; re-enable it
        sbl r0, enableA(r0)

        ;; clear interrupt A
        addi r1, r0, 0xFF
        sbl r1, flagA(r0)

        add r14, r0, r15        ; save the return PC

        ;; destination address, must make sure that
        ;; a valid one is present
        lbu r1, LAPHWAD(r0)
        addi r2, r0, 0xFF
        sne r3, r2, r1
        bnez r3, goodHWAD
        jr r14

goodHWAD:
        addi r1, r0, LAPHWAD
        addi r2, r0, FRAME
        addi r3, r0, 6
        acall r15, MemCopy

        ;; source address
        addi r1, r0, MYHWAD
        addi r2, r0, FSRC
        addi r3, r0, 6
        acall r15, MemCopy

        ;; now set the type of the frame, which is IP
        addi r1, r0, 1
        addi r2, r0, IPN
        sbh r2, FTYPE(r0)
        sbl r2, FTYPE(r1)

        ;; that's all for Ethernet headers; now need to do
        ;; the IP headers

        add r1, r0, r0

        ;; IPV4, len = 20
        addi r2, r0, IPVN
        sbl r2, FEDATA(r1)
        addi r1, r1, 1

        ;; service type is normal precedence
        sbl r0, FEDATA(r1)
        addi r1, r1, 1

        ;; length of our packet is IP hdrs + UDP hdrs + data
        ;; 20 + 8 + 11
        addi r2, r0, 39
```

```
sbh r2, FEDATA(r1)
addi r1, r1, 1
sbl r2, FEDATA(r1)
addi r1, r1, 1

;; IP ident number, we'll use 7
addi r2, r0, 7
sbh r2, FEDATA(r1)
addi r1, r1, 1
sbl r2, FEDATA(r1)
addi r1, r1, 1

;; flags & fragmentation offset
;; none = 0
sbh r0, FEDATA(r1)
addi r1, r1, 1
sbl r0, FEDATA(r1)
addi r1, r1, 1

;; time to live, 30 seconds
addi r2, r0, 30
sbl r2, FEDATA(r1)
addi r1, r1, 1

;; Pcol = UDP
addi r2, r0, UDPN
sbl r2, FEDATA(r1)
addi r1, r1, 1

;; checksum, handled by a different routine
addi r1, r1, 2

;; source address
addi r1, r0, MYIPAD
addi r2, r0, FEDATA
addi r2, r2, IPSRC
addi r3, r0, 4
acall r15, MemCopy

;; destination address
addi r1, r0, LAPIPAD
addi r2, r0, FEDATA
addi r2, r2, IPDST
addi r3, r0, 4
acall r15, MemCopy

;; IP headers are almost done; need to do checksum
addi r1, r0, FEDATA
acall r15, doIPCSum

;; now do the UDP headers
;; UDP source port
addi r1, r0, SRCPORT
addi r2, r0, UDPRCV
sbh r2, FEDATA(r1)
addi r1, r1, 1
sbl r2, FEDATA(r1)
addi r1, r1, 1

;; UDP destination port
```

```
        addi r2, r0, UDPSND
        sbh r2, FEDATA(r1)
        addi r1, r1, 1
        sbl r2, FEDATA(r1)
        addi r1, r1, 1

        ;; Datagram length (UPD headers + data length)
        addi r2, r0, 19
        sbh r2, FEDATA(r1)
        addi r1, r1, 1
        sbl r2, FEDATA(r1)
        addi r1, r1, 1

        ;; don't bother with UDP checksum
        addi r2, r0, 0xFFFF
        sbh r2, FEDATA(r1)
        addi r1, r1, 1
        sbl r2, FEDATA(r1)
        addi r1, r1, 1

        ;; now let's send a message
        addi r1, r0, msgA
        addi r2, r0, FEDATA
        addi r2, r2, UDPDATA
        addi r3, r0, 11
        acall r15, MemCopy

        addi r1, r0, 53
        acall r15, NetSend
        jr r14

        ;; interrupt B handler
        ;; when interrupt B occurs, re-enable the flag
        ;; that allows processing of interrupt A
        ;; this is needed so that we don't flood my laptop
        ;; with UDP packets
intBf:
        ;; re-enable sending on intA
        addi r1, r0, 0xFF
        sbl r1, enableA(r0)

        ;; clear intB
        addi r1, r0, 0xFF
        sbl r1, flagB(r0)
        jr r15

MemCopy:
        ;; this subroutine is crucial to maintaining my sanity
        ;; src address = r1, target address = r2
        ;; length = r3 (in bytes)
        ;; this routine assumes the src and dest addresses
        ;; are within the first 64 K of memory

        lbu r4, 0x0(r1)
        sbl r4, 0x0(r2)
        addi r1, r1, 1
        addi r2, r2, 1
        subi r3, r3, 1
        bnez r3, MemCopy
        jr r15
```

```
NetInit:
        ;; this routine initializes the ethernet interface
        ;; will use MAC address 0800BEEFBEEF

        addi r1, r0, 1

        ;; force a reset on the chip
        addi r2, r0, 0x0114
        addi r3, r0, 0x0055
        iow r2, PPPtr(r0)
        lsri r2, r2, 8
        iow r2, PPPtr(r1)
        iow r3, PPD0(r0)
        lsri r3, r3, 8
        iow r3, PPD0(r1)

wait:   addi r2, r0, 10
        subi r10, r10, 1
        bnez r10, wait

        ;; setup our MAC address
        ;; this is kinda strange - have to byte swap each
        ;; 2-byte pair because the CS8900a is little
        ;; endian
        addi r2, r0, 0x0158
        addi r3, r0, 0x0008
        iow r2, PPPtr(r0)
        lsri r2, r2, 8
        iow r2, PPPtr(r1)
        iow r3, PPD0(r0)
        lsri r3, r3, 8
        iow r3, PPD0(r1)

        addi r2, r0, 0x015A
        addi r3, r0, 0xEFBE
        iow r2, PPPtr(r0)
        lsri r2, r2, 8
        iow r2, PPPtr(r1)
        iow r3, PPD0(r0)
        lsri r3, r3, 8
        iow r3, PPD0(r1)

        addi r2, r0, 0x015C
        addi r3, r0, 0xEFBE
        iow r2, PPPtr(r0)
        lsri r2, r2, 8
        iow r2, PPPtr(r1)
        iow r3, PPD0(r0)
        lsri r3, r3, 8
        iow r3, PPD0(r1)

        ;; setup the RxCtl
        addi r2, r0, 0x0104
        addi r3, r0, 0x0D05
        iow r2, PPPtr(r0)
        lsri r2, r2, 8
        iow r2, PPPtr(r1)
        iow r3, PPD0(r0)
        lsri r3, r3, 8
```

```
        iow r3, PPD0(r1)

        ;; setup the LineCtl
        addi r2, r0, 0x0112
        addi r3, r0, 0x00C0
        iow r2, PPPtr(r0)
        lsri r2, r2, 8
        iow r2, PPPtr(r1)
        iow r3, PPD0(r0)
        lsri r3, r3, 8
        iow r3, PPD0(r1)

        jr r15

NetSend:
        ;; this routine sends data starting at address FRAME
        ;; the length to send is passed via r1
        add r12, r1, r0

        ;; bid for buffer space
        ;; issue the Tx command
        addi r1, r0, 1
        addi r2, r0, 0x00C0
        iow r2, TxCmd(r0)
        lsri r2, r2, 8
        iow r2, TxCmd(r1)

        ;; tell how long the frame is
        iow r12, TxLen(r0)
        lsri r13, r12, 8
        iow r13, TxLen(r1)

        ;; wait until space is available
        addi r2, r0, 0x0138
        iow r2, PPPtr(r0)
        lsri r2, r2, 8
        iow r2, PPPtr(r1)

bidlp:  ior r3, PPD0(r0)        ; don't care about low byte
        ior r3, PPD0(r1)
        andi r3, r3, 0x0001
        beqz r3, bidlp

        ;; now write all the data stored at FRAME
        add r13, r0, r0
txlp:   lbu r2, FRAME(r13)
        iow r2, RxTx0(r0)
        addi r13, r13, 1
        lbu r2, FRAME(r13)
        iow r2, RxTx0(r1)
        addi r13, r13, 1
        sge r4, r13, r12
        beqz r4, txlp

        jr r15

NetRecv:
        ;; for the moment, this routine will poll the ethernet
        ;; adapter and see if there are any frames waiting
        ;; if no frames, return a length of 0 in r1
```

```
        ;; if there is a frame, return its length in r1
        ;; and type in r2
        ;; Supported types are IP=0x0800 and ARP=0x0806

        ;; 1st, poll the RxEvent register and
        ;; check the RxOK bit -- bit #8
        addi r1, r0, 1
        addi r2, r0, 0x0124
        iow r2, PPPtr(r0)
        lsri r2, r2, 8
        iow r2, PPPtr(r1)

        ior r4, PPD0(r0)        ; Low RxEvent byte
        ior r3, PPD0(r1)        ; High RxEvent byte

        andi r3, r3, 0x0001

        ;; if r3 == 1, then we have a frame
        ;; otherwise, just return

        bnez r3, gotFrame
        add r1, r0, r0
        jr r15

gotFrame:
        ;; re-read the status from RxTx0, high byte first
        ;; but we don't really care what the value is
        addi r1, r0, 1
        ior r4, RxTx0(r1)
        ior r3, RxTx0(r0)

        ;; now we read the length of the frame
        ;; and store it in r12

        ior r3, RxTx0(r1)
        ior r2, RxTx0(r0)
        lsli r3, r3, 8
        or r12, r2, r3

        ;; finally, read each byte from the adapter
        ;; and store it somewhere
        add r5, r0, r0
rxlp:   ior r3, RxTx0(r0)
        sbl r3, ETHIN(r5)
        addi r5, r5, 1
        ior r3, RxTx0(r1)
        sbl r3, ETHIN(r5)
        addi r5, r5, 1
        sge r6, r5, r12
        beqz r6, rxlp

        ;; return the size, type in r1, r2
        lbu r2, ITYPE(r0)       ; MSB of type
        lbu r3, ITYPE(r1)       ; LSB of type
        lsli r2, r2, 8
        or r2, r2, r3
        add r1, r0, r12

        jr r15
```

```
ARPFunc:
        ;; this method handles ARP requests
        ;; and ARP responses according to the following
        ;; if ARP request and asking my IP address
        ;; then tell the requester my IP & MAC
        ;; if ARP response sent to me, then clearly
        ;; I want the sender's IP address for a UDP
        ;; packet I'm about to send

        ;; look at the ARP operation
        addi r1, r0, 1
        addi r5, r0, ARPOP
        addi r5, r5, 1                  ; only care about the LSB of the OP
        lbu r6, IEDATA(r5)

        ;; if r6==1, arp request; else arp response
        seq r7, r6, r1
        beqz r7, ARPRes

        ;; okay, we have an ARP request
        ;; look at the target IP and see if it's mine

        ;; 1st IP byte; r2 will be mine, r3 will be target
        add r5, r0, r0
        addi r6, r0, TIPA
        add r4, r0, r0
        addi r8, r0, 4

ARPMatchLoop:
        lbu r2, MYIPAD(r5)
        lbu r3, IEDATA(r6)
        seq r7, r2, r3

        bnez r7, ARPReqMatch
        jr r15                          ; IP address isn't mine

ARPReqMatch:

        addi r5, r5, 1                  ; increment to check the next byte
        addi r6, r6, 1
        addi r4, r4, 1                  ; inc the loop counter
        seq r7, r4, r8
        beqz r7, ARPMatchLoop

        ;; if we got this far, then the target IP address is mine
        ;; next on the list is to prepare our ARP
        ;; response

        ;; copy sender's MAC to DEST field of outgoing frame
        addi r1, r0, IEDATA
        addi r1, r1, SHWA
        addi r2, r0, FRAME
        addi r3, r0, 6
        add r14, r0, r15        ; save the return PC
        acall r15, MemCopy

        ;; copy my MAC to the sender field of FRAME
        addi r1, r0, MYHWAD
        addi r2, r0, FSRC
        addi r3, r0, 6
```

```
acall r15, MemCopy

;; now set the type of the frame, which is ARPN
addi r1, r0, 1
addi r2, r0, ARPN
sbh r2, FTYPE(r0)
sbl r2, FTYPE(r1)

;; write the ARP headers to FRAME
add r1, r0, r0
addi r2, r0, 1                ; the HW type is Ethernet, 0x0001
sbh r2, FEDATA(r1)
addi r1, r1, 1
sbl r2, FEDATA(r1)
addi r1, r1, 1

addi r2, r0, IPN        ; the Proto is IP
sbh r2, FEDATA(r1)
addi r1, r1, 1
sbl r2, FEDATA(r1)
addi r1, r1, 1

addi r2, r0, 6                ; HW address length is 6
sbl r2, FEDATA(r1)
addi r1, r1, 1

addi r2, r0, 4                ; IPv4 address length is 4
sbl r2, FEDATA(r1)
addi r1, r1, 1

;; write the ARP response operation, which is 0x0002
addi r1, r0, 0x0002
addi r2, r0, ARPOP
sbh r1, FEDATA(r2)
addi r2, r2, 1
sbl r1, FEDATA(r2)

;; write my HW and IP addresses to FRAME
;; this data can be copied from MYHWAD, MYIPAD (contiguous)
addi r1, r0, MYHWAD
addi r2, r0, FEDATA
addi r2, r2, SHWA
addi r3, r0, 10
acall r15, MemCopy

;; finally write the sender's HW and IP addresses
;; to FRAME; this data is stored in
;; ETHIN + SHWA
addi r1, r0, IEDATA
addi r1, r1, SHWA
addi r2, r0, FEDATA
addi r2, r2, THWA
addi r3, r0, 10
acall r15, MemCopy

;; now send the frame
addi r1, r0, ARPLEN
acall r15, NetSend

jr r14
```

```
ARPRes:
        ;; here's where I handle ARP responses
        ;; the first thing to do is make sure that
        ;; my HW address is in the target field

        ;; r2 will be mine, r3 will be target
        add r5, r0, r0
        addi r6, r0, THWA
        add r4, r0, r0
        addi r8, r0, 6

ARPMatchLoop2:
        lbu r2, MYHWAD(r5)
        lbu r3, IEDATA(r6)
        seq r7, r2, r3

        bnez r7, ARPResMatch
        jr r15                          ; MAC address isn't mine

ARPResMatch:

        addi r5, r5, 1                  ; increment to check the next byte
        addi r6, r6, 1
        addi r4, r4, 1                  ; inc the loop counter
        seq r7, r4, r8
        beqz r7, ARPMatchLoop2

        ;; if we got this far, then the ARP response was
        ;; meant for me; copy sender HW address of the response
        ;; to LAPHWAD

        addi r1, r0, IEDATA
        addi r1, r1, SHWA
        addi r2, r0, LAPHWAD
        add r14, r15, r0
        acall r15, MemCopy

        ;; don't need to do anything else
        ;; just return
        jr r14

ARPReq:
        ;; this routine is used to make an ARP request
        ;; the actual response from the network is handled
        ;; inside of ARPFunc

        add r14, r0, r15        ; save the return PC

        ;; broadcast to DEST field of outgoing frame
        addi r3, r0, 6
        addi r2, r0, 0xFFFF
        add r1, r0, r0

bcastlp:
        sbh r2, FDEST(r1)
        addi r1, r1, 1
        subi r3, r3, 1
        bnez r3, bcastlp
```

```
        ;; copy my MAC to the sender field of FRAME
        addi r1, r0, MYHWAD
        addi r2, r0, FSRC
        addi r3, r0, 6
        acall r15, MemCopy

        ;; now set the type of the frame, which is ARPN
        addi r1, r0, 1
        addi r2, r0, ARPN
        sbh r2, FTYPE(r0)
        sbl r2, FTYPE(r1)

        ;; write the ARP headers to FRAME
        add r1, r0, r0
        addi r2, r0, 1                ; the HW type is Ethernet, 0x0001
        sbh r2, FEDATA(r1)
        addi r1, r1, 1
        sbl r2, FEDATA(r1)
        addi r1, r1, 1

        addi r2, r0, IPN        ; the Proto is IP
        sbh r2, FEDATA(r1)
        addi r1, r1, 1
        sbl r2, FEDATA(r1)
        addi r1, r1, 1

        addi r2, r0, 6                ; HW address length is 6
        sbl r2, FEDATA(r1)
        addi r1, r1, 1

        addi r2, r0, 4                ; IPv4 address length is 4
        sbl r2, FEDATA(r1)
        addi r1, r1, 1

        ;; write the ARP request operation, which is 0x0001
        addi r1, r0, 0x0001
        addi r2, r0, ARPOP
        sbh r1, FEDATA(r2)
        addi r2, r2, 1
        sbl r1, FEDATA(r2)

        ;; write my HW and IP addresses to the sender field
        ;; this data can be copied from MYHWAD, MYIPAD (contiguous)
        addi r1, r0, MYHWAD
        addi r2, r0, FEDATA
        addi r2, r2, SHWA
        addi r3, r0, 10
        acall r15, MemCopy

        ;; finally write 0's for the target HW address
        ;; and my laptop as the target IP address
        addi r3, r0, 6
        addi r1, r0, THWA

zeroMAClp:
        sbh r0, FEDATA(r1)
        addi r1, r1, 1
        subi r3, r3, 1
        bnez r3, zeroMAClp
```

```
        addi r1, r0, LAPIPAD
        addi r2, r0, FEDATA
        addi r2, r2, TIPA
        addi r3, r0, 4
        acall r15, MemCopy

        ;; now send the frame
        addi r1, r0, ARPLEN
        acall r15, NetSend

        jr r14

IPFunc:
        ;; this routine handles incoming UDP packets as such:
        ;; verify dest IP and incoming port #
        ;; get first byte of UDP data, display on LED

        ;; make sure dest IP address is 192.168.0.2
        add r5, r0, r0                ; my IP address
        addi r6, r0, IPDST        ; IP in question
        add r4, r0, r0
        addi r8, r0, 4

IPLoop:
        lbu r2, MYIPAD(r5)
        lbu r3, IEDATA(r6)
        seq r7, r2, r3

        bnez r7, IPMatch
        jr r15                        ; IP address isn't mine

IPMatch:
        addi r5, r5, 1                ; increment to check the next byte
        addi r6, r6, 1
        addi r4, r4, 1                ; inc the loop counter
        seq r7, r4, r8
        beqz r7, IPLoop

        ;; the IP address is correct, now check for UDP protocol
        addi r1, r0, UDPN
        addi r2, r0, PCOL
        lbu r3, IEDATA(r2)
        seq r4, r1, r3
        bnez r4, isUDP
        jr r15

isUDP:
        ;; it is a UDP packet; so check for the proper port
        ;; on which we are listening
        addi r1, r0, UDPRCV
        addi r2, r0, DSTPORT
        lbu r3, IEDATA(r2)
        lsli r3, r3, 8
        addi r2, r2, 1
        lbu r4, IEDATA(r2)
        or r3, r3, r4
        seq r4, r1, r3
        bnez r4, goodUDP
        jr r15
```

```
goodUDP:
        ;; ok, we have the UDP packet, we have now only to grab a byte
        ;; and display it plus the number read on PIN

        addi r1, r0, UDPDATA
        lbu r2, IEDATA(r1)

        ;; strobe the input port
        sbl r0, PIN(r0)                 ; strobe a value into PIN
        lbu r3, PIN(r0)                 ; load PIN to r3
        lsri r3, r3, 3                 ; r3 = r3 >> 3
        add r2, r2, r3
        add r14, r15, r0
        acall r15, led
        jr r14

doIPCSum:
        ;; this routine calculates the IP checksum
        ;; input: r1 is address of IP header
        ;; output: writes checksum to the IP header
        ;; assumes all other IP information already
        ;; in the header

        ;; store the checksum in r2, use r3 as a counter
        add r2, r0, r0
        addi r3, r0, 20
        add r7, r0, r1                  ; backup original address

        ;; need to clear the checksum memory first
        addi r6, r1, IPCSUM
        sbl r0, 0x0(r6)
        sbl r0, 0x1(r6)

IPCSumLoop:
        lbu r5, 0x0(r7)
        lbu r6, 0x1(r7)
        lsli r5, r5, 8
        or r5, r5, r6
        addi r7, r7, 2
        addc r6, r2, r5
        add r2, r2, r5
        add r2, r2, r6
        subi r3, r3, 2
        bnez r3, IPCSumLoop

        ;; create the 1's compliment of the result
        xori r2, r2, 0xFFFF

        ;; write the checksum to memory
        addi r6, r1, IPCSUM
        sbh r2, 0x0(r6)
        sbl r2, 0x1(r6)

        jr r15

        ;; ------------------------------------
        ;; data memory starts here
        ;; ------------------------------------

        ;; define 7 segment values
```

```
ledda:  .db         0x77                ; 0
        .db         0x12                ; 1
        .db         0x5D                ; 2
        .db         0x5B                ; 3
        .db         0x3A                ; 4
        .db         0x6B                ; 5
        .db         0x6F                ; 6
        .db         0x52                ; 7
        .db         0x7F                ; 8
        .db         0x7B                ; 9
        .db         0x7E                ; A
        .db         0x2F                ; B
        .db         0x65                ; C
        .db         0x1F                ; D
        .db         0x6D                ; E
        .db         0x6C                ; F

        ;; flags that get cleared when an interrupt occurs
        ;; this is a hack because I have neither the
        ;; time nor the desire to implement a stack
        ;; on this demonstration program
flagA:
        .db         0xFF

flagB:
        .db         0xFF

        ;; interrupt A get disabled after occuring until
        ;; an interrupt on B occurs
enableA:
        .db         0x0

msgA:
        ;; message sent when interrupt A occurs
        .db         105
        .db         110
        .db         116
        .db         101
        .db         114
        .db         114
        .db         117
        .db         112
        .db         116
        .db         32
        .db         65

MYHWAD:
        ;; my MAC address
        .db         0x08
        .db         0x00
        .db         0xBE
        .db         0xEF
        .db         0xBE
        .db         0xEF

MYIPAD:
        ;; my IP address
        .db         192
        .db         168
        .db         0
```

```
        .db         2

LAPIPAD:
        ;; laptop's IP address
        .db         192
        .db         168
        .db         0
        .db         1

LAPHWAD:
        ;; laptop's MAC address
        .db         0xFF
        .db         0x00
        .db         0x00
        .db         0x00
        .db         0x00
        .db         0x00

        ;; outgoing Ethernet frame needs to go here
FRAME:
FDEST:      .ds         6
FSRC:       .ds         6
FTYPE:      .ds         2
FEDATA:     .ds         1504

        ;; store the incoming frame here
ETHIN:
IDEST:      .ds         6
ISRC:       .ds         6
ITYPE:      .ds         2
IEDATA:     .ds         1504

        .end
```

Appendix C

Project Files and Source Code Availability

This source code referenced in this thesis is available for download from the World Wide Web:

`http://www.lehigh.edu/~jrp2/emrisc16.html`

The files available include schematics, VHDL code, Xilinx Foundation 2.1i project files, the EmRISC16 Assembler source, the demonstration assembly program, and this document.

Appendix D

Vita

James "J. R." Petrus was born in Butler, Pennsylvania on April 20, 1979 to James and Janet Petrus. He entered Lehigh University in Fall 1997 and graduated summa cum laude with a Bachelor of Science degree in computer engineering in June 2001. Named a President's Scholar upon graduation, James accepted this award and continued his studies at Lehigh University. He is currently working towards his Master of Science in computer engineering, and expects to graduate in May 2003. During his graduate years at Lehigh, James served as a teaching assistant for the Computer Science and Engineering Department as well as a research assistant in Lehigh's Vision and Software Technology Lab.

END OF TITLE