Design and Evaluation of an FPGA Based Microprocessor Project Board

Ross Brennan
BAI Computer and Electronic Engineering

Final Year Project May 2003

Supervisor: Michael Manzke

Acknowledgements[1]

I would like to thank the Technicians in the Computer Science Department for helping me out with supplies and tools when I needed them and to my Uncle Pat, who gave me advice on methods of interfacing different signalling standards together. Special thanks go to Tom Kearney, whose help in acquiring components and advice during the hardware design phase of the project was invaluable, without his help I probably wouldn't have gotten the project hardware up and running. Finally, I'd like to thank my supervisor, Michael Manzke, for giving me the oportunity to work on this project and for his support and guidance throughout the year.

[1]This document has been fully typeset in LaTeX

Contents

1 Introduction **1**
1.1 The Current Design Project 1
1.1.1 The Motorola MC68008 2
1.1.2 Project Components 4
1.1.3 Project Tasks 4
1.2 Motivation 5
1.3 An FPGA Based Design 5
1.4 Platform Options 5
1.5 Initial Testing 6

2 The LEON Core **8**
2.1 Origins 8
2.2 Model Architecture 8
2.2.1 Integer Unit 9
2.2.2 Caches 9
2.2.3 AMBA AHB/APB Bus 10
2.2.4 Debug Support Unit 12
2.2.5 Memory Controller 12
2.2.6 Timers 12
2.2.7 Internal UARTs 13
2.2.8 IRQ Controllers 13
2.2.9 Parallel I/O Port 13
2.2.10 PCI Interface 13
2.3 Signalling 14
2.4 Configuration 16
2.5 Simulation 17
2.6 Synthesis 19

3 Modifying the LEON Core **20**
3.1 Identifying Project Tasks 20
3.2 VHDL Model Heirarchy 22
3.3 Clock Speed 22
3.3.1 Simulation 23
3.4 Caches 23
3.4.1 Alterations to the Configuration Records 23

3.4.2 Alterations to the Source Code 24
3.4.3 Simulation 25
3.5 Internal UARTs 25
3.5.1 Alterations to the Configuration Records 25
3.5.2 Alterations to the Source Code 26
3.5.3 Simulation 27
3.6 Reset Generation 27
3.6.1 Alterations to the Configuration Records 27
3.6.2 Alterations to the Source Code 27
3.6.3 Simulation 28
3.7 Memory Map and Controller 28
3.7.1 Alterations to the Configuration Records 29
3.7.2 Alterations to the Source Code 29
3.7.3 Simulation 32
3.8 Model Synthesis 32
3.8.1 Configuration 32
3.8.2 Synthesis 33
3.8.3 Constraints 34
3.8.4 Mapping, Placing and Routing 34
3.8.5 Bitfile Generation 35
3.9 Downloading the Core 35

4 Hardware Design and Assembly 36
4.1 Control Logic 37
4.1.1 Clock Divider 38
4.1.2 Reset Generator 38
4.1.3 Memory Controller 39
4.1.4 Simulation 41
4.2 Hardware Decisions 42
4.2.1 TTL vs LVTTL 42
4.2.2 TSOP vs DIP 42
4.2.3 Interconnects 42
4.3 ROM 43
4.3.1 Programming the ROM 44
4.4 RAM 45
4.5 UARTs 46
4.5.1 LVTTL to RS232 46
4.6 Final Design 47

5 Testing and Conclusions 49
5.1 Software Tools 49
5.1.1 The LECCS Compiler 49
5.1.2 The TSIM LEON Simulator 50
5.1.3 DSUMON 50
5.2 The Infinite Loop Test Program 51

5.2.1 Test Results 52
5.3 Conclusions 53
5.4 Future Work 53

A Demo Program **54**

B LEON **56**
B.1 LEON Configuration File 56
B.2 Pin Constraints 57
B.3 Project Board Circuit Diagram 58

Bibliography **60**

List of Figures

1.1 Block Diagram of the MC68008 Microprocessor 2
1.2 Simplified MC68008 Memory Read Cycle 3
1.3 Download Interface Cable 7

2.1 Block Diagram of the LEON model architecure 9
2.2 Memory Controller Interface Signals 12
2.3 Block Diagram of the LEON Microprocessor 14
2.4 Simplified 32-bit LEON Memory Read Cycle 15
2.5 Simplified 8-bit LEON Memory Read Cycle 15
2.6 Waveform results of standard tb_func8 testbench 18

3.1 Layout of the LEON Core 22
3.2 State flow diagram for updated LEON memory cycle 31

4.1 Block Diagram of the Proposed Project Hardware 36
4.2 Block Diagram of the Control Logic 37
4.3 Diagram of the clock divider output 38
4.4 Diagram of Reset Signal Generation 39
4.5 Block Diagram of the ROM chip (DIP adapter fitted) 43
4.6 Block Diagram of the RAM chip (DIP adapter fitted) 45
4.7 Block Diagram of the RS232 Interface Circuitry 46
4.8 Front View of Completed Hardware 47
4.9 Rear View of Completed Hardware 48

5.1 Expected Waveform Results of Infinite Loop Program 52
5.2 Actual Waveform Trace from Logic Analyser 52

B.1 Circuit Diagram of the Project Hardware System 59

List of Tables

1.1 Signal Classifications of the MC68008 3
1.2 Prototyping Board Configuration Modes 6

2.1 Default AHB Memory Map 10
2.2 On-chip Registers 11
2.3 Parallel Port Usage (8/16-bit Mode) 13
2.4 Signal Classifications of the LEON processor 14

3.1 Project Memory Map 28

4.1 Device Memory Map 39
4.2 Pin Mappings for Convertor 44
4.3 Colour coding of wire-wrap connections 48

A.1 Pin Constraints used for the Demo Program 55

B.1 User Constraints Pin Allocations 58

Abstract

The goal of this project was to evaluate the suitability of modifying an open source VHDL implementation of a RISC microprocessor with the aim of using it as a teaching aid aimed at second year computer-science and third year computer-engineering students.
The core that was evaluated is the LEON SPARC-V8 compatible processor model, which was released under the GNU-LGPL and is freely available to download from the internet.
The possibility of upgrading the current microprocessor design project by using the LEON core in place of the current Motorolla M68008 microprocessor was also explored and a prototype project board was designed and assembled in order to test the operation of the modified processor model.
This report outlines the changes made to the LEON model in order to make it suitable for integration into a new design project, based on the format of the original one. The model is tested for compatibility and several test programs, written in RISC assembly code, are evaluated on the processor model.

Chapter 1

Introduction

This project set out to evaluate the suitability of modifying an FPGA[1] based microprocessor with the intention of using it as a teaching-aid and, if successful, exploring the possibility of upgrading the existing microprocessor design project, which is undertaken by Computer Science and Computer Engineering students, using FPGA processor instead of the current Motorola processor.
The model that was assessed as the replacement processor for the project is the LEON2-1.0.10-xst [2] synthesisable VHDL RISC microprocessor core.
The LEON processor would only be suitable as an upgrade if it could be integrated into a project format similar to that of the current design project. It should replace all of the functionallity of the Motorola processor while still providing a solid base for students to acquire a good working knowledge of the steps required to interface a microprocessor with peripheral devices, as well as understanding and being able to write machine assembly code.
This chapter describes the current microprocessor project and sets out the arguments for and against upgrading the project to be based around a synthesisable processor model as opposed to the Motorola processor.

1.1 The Current Design Project

The current microprocessor design project is based on the Motorola MC68008 CISC microprocessor and is targeted at second year Computer Science and third year Computer Engineering students. The main aim of the project is to introduce students to the operation of microprocessor systems at a fundamental level and to give them "hands-on" experience in the construction of these systems.
The design project sets out specific tasks for the students to work to, over a six week period, with each task building on the previous. Successful completion of all of the tasks results in a fully functional microprocessor system comprising of one EPROM, two RAM chips and two serial ports. The system is designed to be basic enough to allow completion

[1]Field Programmable Gate Array
[2]Available under the GNU-LGPL from http://www.gaisler.com

within the specified timeframe while, at the same time, enabling the students to learn about the operation and interaction of the hardware involved in creating the system. The following list of suggested goals is recommended to students undertaking the current desing project:

- WEEK 1: Verify the operation of the on-board clock circuitry and learn how to programme the Gate Array Logic (GAL) chips
- WEEK 2: Implement the core processor architecture (Clock, GALs, EPROM) and verify that the processor can read from the EPROM
- WEEK 3: Add and verify the RAMs, R6551s and MAX232 then implement and test a transparent-link program
- WEEK 4: Write and implement a polled monitor program and implement bus timeouts
- WEEK 5: Extra week to finish project work and start project write-up
- WEEK 6: Hand up project report and completed hardware

1.1.1 The Motorola MC68008

The original Motorola MC68000 family of microprocessors was introduced in 1982, with the MC68008 being released shortly afterwards. The MC68008 has an internal 16-bit architecture with an 8-bit external data bus and a 20-bit address bus (1MB address space). It is a CISC processor and has no internal caches, memory management unit (MMU) or floating point unit (FPU), however these may be added optionally as external devices, enhancing the operation of the processor.

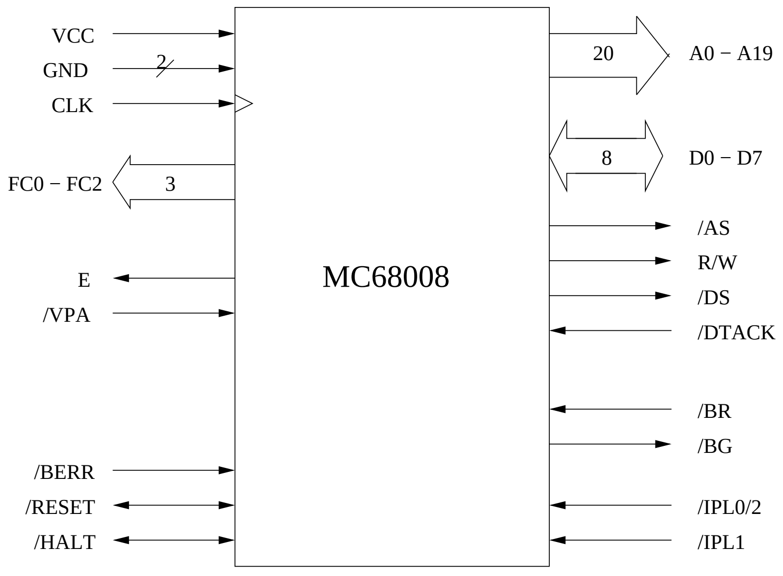

Figure 1.1: Block Diagram of the MC68008 Microprocessor

The various MC68008 signals are summarized in table 1.1. All of these signals operate using 5V transistor-to-transistor logic (TTL).

Signal Type	Label	No.
Power and Timing	Vcc, Gnd(2), Clk	4
Processor Status	FC0, FC1, FC2	3
M6800 Peripheral Ctrl	E, /VPA	2
System Control	/BERR, /RESET, /HALT	3
Address Bus	A0 - A19	20
Data Bus	D0 - D7	8
Asynch Bus Control	/AS, R/W, /DS, /DTACK	4
Bus Arbitration Ctrl	/BR, /BG	2
Interrupt Ctrl	IPL0/2, /IPL1	2
TOTAL:		48

Table 1.1: Signal Classifications of the MC68008

MC68008 Memory Accesses

The Motorola MC68008 uses an asynchronous bus transfer protocol with a transfer rate of 1 byte per access. Figure 1.2 outlines a basic read cycle performed by the processor. The chip and output enable signalling is generated by external circuitry using the address and data strobes combined with the memory address on the bus and whether or not the cycle is a read or a write cycle.

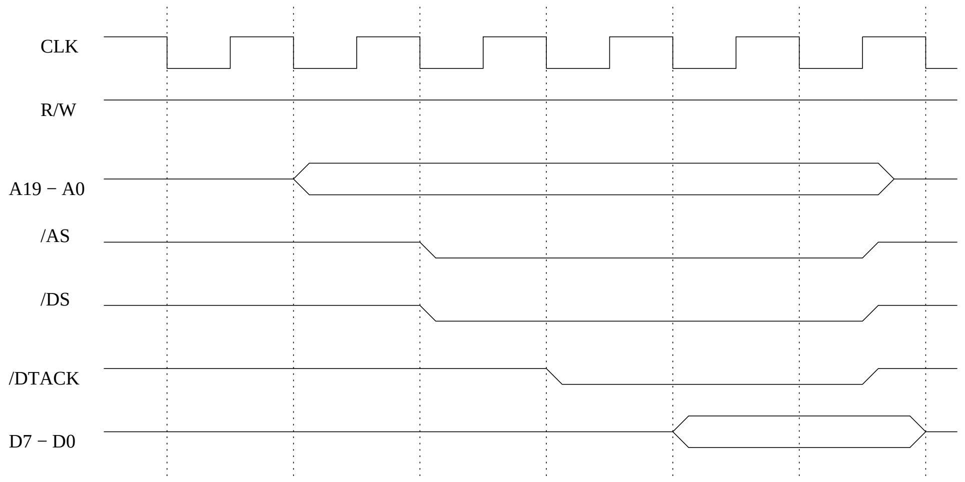

Figure 1.2: Simplified MC68008 Memory Read Cycle

The /DTACK signal is used to notify the processor that there is valid data on the data bus and that the memory cycle may be terminated on the next clock edge. Memory accesses can be delayed by delaying the assertion of the /DTACK signal. This may be necessary for slow external memory devices or when performing memory mapped I/O.

1.1.2 Project Components

The components used in the project consist of one EPROM, two RAM chips, two GALs, two ACIAs and one MAX232. The EPROM has an 8-bit data bus and a capacity of 8kB. The RAM chips also have 8-bit data buses and a capacity of 2kB each. These chips are controlled using a combination of the MC68008's asynchronous bus control signals combined with the control signals generated in the GALs.
The GALs contain a 4-bit clock divider as well as logic that controls the data transfer acknowledge, output enable, read/write and chip enable signals.
The Asynchronous Communication Interface Adapters (ACIAs) communicate with the processor using the M6800 peripheral interface via the "E" and "/VPA" control signals.
The MAX232 chip is used to translate the voltage levels between TTL and RS232 standards in order that the system can be connected to a standard serial port.

1.1.3 Project Tasks

- The first task is to verify that the on-board clock circuitry is operational and generating a 15MHz signal. This is tested by connecting the signal to an oscilliscope and measuring the period of the resultant waveform.

- A 4-bit counter is then designed and implemented within one of the GALs. This counter is used to divide the 15MHz signal down to 8MHz and 1MHz, which are used to drive the processor and the ACIAs respectively.

- The reset circuitry is designed to debounce the signal from the reset push-button and to delay the reset signal by holding it at a logic-low level for at least three clock cycles. This ensures that the processor and peripheral components have reset correctly.

- The MC68008's 1MB address space is then segmented into regions where the ROM, RAM and ACIAs are to be mapped. This memory-mapping is then used to generate the enable signals, for the chips, from logic within the GAL.

- At this stage, the EPROM is programmed with an infinite-loop test program. This allows the students to verify that the control logic is working and that the processor is reading information from the ROM. Trace information is captured using a logic state analyser connected to the address and data buses.

- The two ACIAs are then connected to the processor and the MAX232. The operation of serial ports are tested using a transparent link program, which allows two computers to be connected together using a hyperterm session.

- At this stage, the project hardware has been completed and the students have built a fully functional microprocessor system. A monitor program is then implemented and downloaded onto the EPROM. This is the final task in the completion of the microprocessor design project.

1.2 Motivation

The main reasons for upgrading the design project are to take advantage of new technology advances both in processor architecture and electronic technology. The current design project is inflexible and based on ageing technology and, as a result, it is becoming more difficult and expensive to replace faulty components.
The current project is based on a 20 year old 16-bit CISC microprocessor and while this serves to teach students about how a processor interacts with peripherals it does not allow them to learn about the architecture of current RISC processor technology, which is used in the majority of microprocessors in operation today. The TTL signalling standards used in the current project have also been superceded by new LVTTL standards that operate at 3.3V instead of 5V, leading to more power efficient designs.

1.3 An FPGA Based Design

It was decided to upgrade the project to be based on an FPGA solution as this provided the most versatility in design. Due to the fact that FPGAs are configurable, any future project design would not have to be constrained to a particular chip type. There are plenty of open-source models of processors designed around different architectures and this was seen as one of the major advantages of using an FPGA, as it meant that any new design project could be easily based around any type of processor that had been implemented using a hardware description language such as VHDL, Verilog or Handel-C.
Using custom hardware to aid in the teaching of computer architecture is not a new concept and its success has already been demonstrated in several different projects, for example the work carried out to design and implement custom hardware and simulation tools at the University of Waikato[1]. Efforts have also been made to design processor architectures from scratch[2] with the intention of providing a simple yet functional platform through which to introduce students to the operation of a microprocessor core. These implementations do not, however, allow the system to grow in complexity as the students understanding of processor design and concepts increase, without major updates to the processor model. By using a fully functional configurable processor model, the complexity of operation may be tailored to suit the needs of different student groups. For this reason, it was decided to base the new design project around a mature, well tested model of a full standardized processor architecture, instead of creating or using an architecture designed solely for the purposes of teaching.
One such option is the LEON processor. This is an open source implementation of a SPARC V8 compliant 32-bit RISC processor. It was chosen to be the target upgrade processor mainly due to the fact that it was highly configurable and open-source, meaning that it could be readily modified to suit the needs of the project.

1.4 Platform Options

The FPGA used in this project is the VirtexII XC2V1000[3] on the VirtexII prototyping board[4]. This was used as the VirtexII chip has a large capacity and is easily

programmed. When used in conjunction with the prototyping board it led to a versatile platform for testing the required circuitry. A Xilinx XC18V04 PROM[5] was used to automatically configure the FPGA with the bit-file.
Some peripheral components were also required to interface with the processor and these were chosen based on hardware compatibility with the Virtex II FPGA. LVTTL compatible PEROM and SRAM chips were used and a MAX3232 was used to convert voltage levels between LVTTL and RS232 standard for the serial port connections. All of these components would be wired-wrapped together.

1.5 Initial Testing

Before any work was done on the LEON core, it was important to test and verify that the hardware, that would be used in the project, was operational. This was important, firstly to ensure that all of the components were functional and secondly, to provide valuable experience in how to generate bit-files targeted at the correct FPGA platform and successfully download and run them on the prototyping board.
The *Xilinx iMPACT* software and the *Xilinx "Parallel Cable IV"* were used to transfer the program bit-file to the FPGA. The first task to be done was to make a connector that was able to interface the parallel cable with the prototyping board.
The prototyping board provided for eight different configuration methods which could be selected using a rotary switch on the board:

Switch Position	Configuration Mode
0	Master Serial PROM
1	Master Serial Upstream
2	Master Select Map PROM
3	Master Select Map Upstream
4	Slave Serial
5	JTAG
6	Select Map
7	External

Table 1.2: Prototyping Board Configuration Modes

It was possible to configure the FPGA directly from the computer using either Slave Serial or JTAG mode, however it was only possible to configure the on-board PROM using JTAG mode so the interface cable was designed to support both JTAG and Slave Serial modes.
A test program was then synthesised for the XC2V1000-FG256-5 and downloaded, using JTAG mode, onto the FPGA. This program was designed to flash the LEDs on the prototyping boardin continuously sequence and in doing so demonstrated that the bit-file had been downloaded correctly and that the programme was running successfully on the FGPA. The demonstration program used is detailed in appendix A.

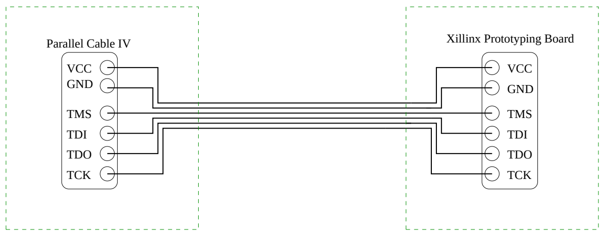

Figure 1.3: Download Interface Cable

The same program was then formatted for use with the XC18V04 PROM and downloaded onto it using the same procedure as for the FPGA. Configuring the PROM instead of the FPGA directly meant that the system would not have to be reprogrammed every time the board was powered-up, as the FPGA would automatically be programmed with the contents of the PROM. This procedure would also verify that the configuration PROM was communicating with the FPGA.
These two tests produced positive results, proving that the basic hardware aspect of the project had been set up and was working correctly. They also gave valuable insight into the operation of the prototyping board as well as the download software and hardware that were required to configure the FPGA and PROM.

Chapter 2

The LEON Core

LEON-P1754 is a VHDL model of a 32-bit processor conforming to the IEEE-1745 standard, which is fully compatible with the SPARC V8 architecture[6]. The model is fully synthesisable and can be implemented on both FPGAs and ASICs. The model incorporates an integer unit, separate instruction and data caches and several peripheral modules, which are connected to the processor through an on-chip AMBA bus.
LEON is provided under the GNU GPL and LGPL. The LGPL applies to the model itself while the remaining support and test files are provided under the GPL. This means that additional modules may be added to the core without being open-source as long as any changes that were made to the model itself remain open.
This chapter describes the architecture and implementation of the unmodified LEON2-1.0.10-xst model [7], which was evaluated as a possible upgrade to the current microprocessor design project. The modifications made to the model in order to make suitable as a teaching aid for the purposes of the upgraded microprocessor design project are discussed in chapter 3.

2.1 Origins

LEON was originally designed by Jiri Gaisler while working for the European Space Administration (ESA) and is currently maintained by Gaisler Research. It was designed for embedded applications with the intention of being used in future satellite systems which are under development by the ESA. A fault-tolerent version of the Leon core that incorporates hardware features capable of withstanding single-event upset errors without loss of data is also available, however it is not open-source. The first release of the LEON core was made available in October 1999, with continuous enhancement and upgrades being released since then.

2.2 Model Architecture

As LEON was designed for embedded applications, many peripheral modules are included in its design. These modules are connected to the processor using two internal buses and provide most of the on-chip functionality. Figure 2.1 outlines the main architectural features of the LEON processor core.

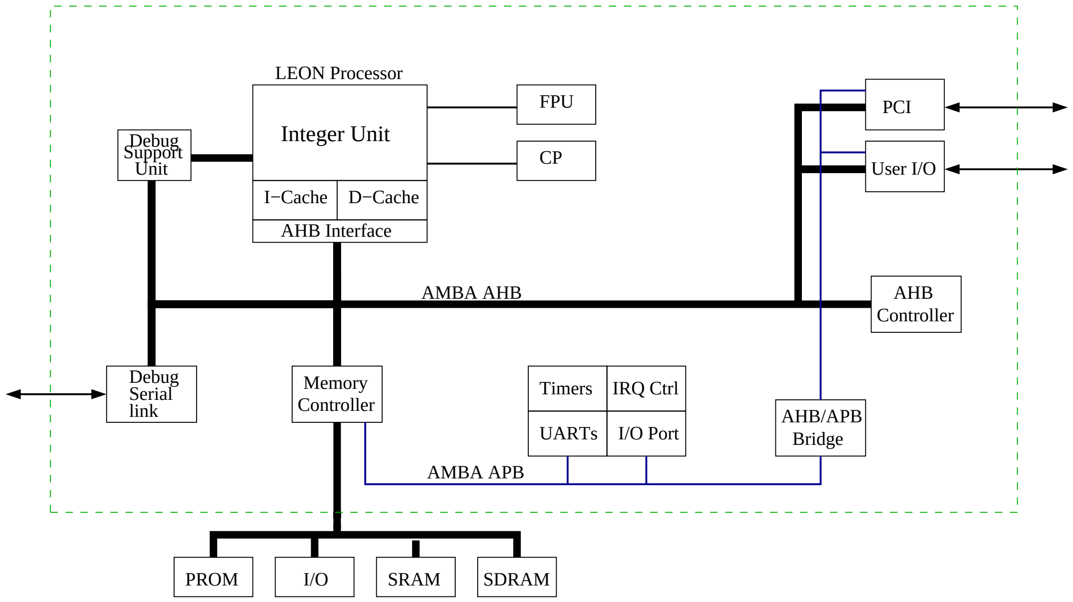

Figure 2.1: Block Diagram of the LEON model architecure

2.2.1 Integer Unit

The integer unit implements the full SPARC V8 standard, including all multiply and divide instructions, and has been certified by SPARC as a fully complient implementation of the standard. The number of register windows is configurable from 2 - 32, with a default setting of 8. The integer unit provides interfaces for an optional Floating Point Unit (FPU) and Coprocessor (CP).
It uses a 5-stage instruction pipeline:

1. *FE* – Instruction Fetch Stage
2. *DE* – Instruction Decode Stage
3. *EX* – Execute Stage
4. *ME* – Memory Stage
5. *WR* – Write Stage

2.2.2 Caches

Seperate instruction and data caches are present within the model. They are connected directly to the integer unit and access the memory controller via the AHB bus. The data cache can perform bus snooping on the AHB bus. Both the instruction and data

caches may be configured individually within the model, each cache having a size of between 1kB and 64kB with a line size of between 4 and 8 words per line. As the size of the caches is increased, so too will the performance, however the overall size of the core will also increase.
Cache sets may be replaced using a pseudo random, least recently replaced (LRR) or least recently used (LRU) algorithm. The LRU scheme has the best performance but also the highest overhead. It is also possible to configure the caches to use line locking, however this will increase the size of the tags.
A cacheability table within the model defines which areas of the address space are to be considered suitable for the instruction and data caches to cache. It defines only the ROM and RAM areas to be cacheable.

```
function is_cacheable(haddr : std_logic_vector(31 downto 24))
return std_logic is variable hcache : std_logic;
begin
    if(haddr(31) = ’0’) and (haddr(30 downto 29) /= "01") then
        hcache := ’1’;
    else
        hcache := ’0’;
    end if;
return(hcache);
```

2.2.3 AMBA AHB/APB Bus

LEON contains full on-chip implementation of the AMBA Advanced Highspeed Bus (AHB) and Advanced Peripheral Bus (APB)[8]. All of the peripheral modules within the LEON core implement the AHB/APB interface, making it easy to add new or remove existing modules. The APB bus is used to access the on-chip registers, while the AHB bus is used for high-speed data transfers. The default address allocations of devices on the AHB are given in table 2.1

Address Range	Size	Mapping	Module
0x00000000 - 0x1FFFFFFF	512 MB	PROM	Memory Controller
0x20000000 - 0x3FFFFFFF	512 MB	I/O	
0x40000000 - 0x7FFFFFFF	1 GB	RAM	
0x80000000 - 0x8FFFFFFF	256 MB	On-chip regs	APB Bridge
0x90000000 - 0x9FFFFFFF	256 MB	Debug Support	DSU

Table 2.1: Default AHB Memory Map

The on-chip registers are laid out according to table 2.2 and are accessible via the APB bridge.

Address	Register	Address	Register
0x80000000	Memory Configuration 1	0x80000090	Interrupt Mask and Priority
0x80000004	Memory Configuration 2	0x80000094	Interrupt Pending
0x80000008	Memory Configuration 3	0x80000098	Interrupt Force
0x8000000C	AHB Failing Address	0x8000009C	Interrupt Clear
0x80000010	AHB Status	0x800000A0	I/O Port Input/Output
0x80000014	Cache Control	0x800000A4	I/O Port Direction
0x80000018	Power-down	0x800000A8	I/O Port Interrupt
0x8000001C	Write Protection 1	0x800000B0	Secondary Interrupt Mask
0x80000020	Write Protection 2	0x800000B4	Secondary Interrupt Pending
0x80000024	LEON Configuration	0x800000B8	Secondary Interrupt Status
0x80000040	Timer 1 Counter	0x800000BC	Secondary Interrupt Control
0x80000044	Timer 1 Reload	0x800000C4	DSU UART Status
0x80000048	Timer 1 Control	0x800000C8	DSU UART Control
0x8000004C	Watchdog	0x800000CC	DSU UART Scaler
0x80000050	Timer 2 Counter		
0x80000054	Timer 2 Reload		
0x80000058	Timer 2 Control		
0x80000060	Scaler Counter		
0x80000064	Scaler Reload		
0x80000070	UART 1 Data		
0x80000074	UART 1 Status		
0x80000078	UART 1 Control		
0x8000007C	UART 1 Scaler		
0x80000080	UART 2 Data		
0x80000084	UART 2 Status		
0x80000088	UART 2 Control		
0x8000008C	UART 2 Scaler		

Table 2.2: On-chip Registers

2.2.4 Debug Support Unit

The debug support unit (DSU) allows non-intrusive debugging of the LEON processor on target hardware. It provides access to all on-chip registers as well as containing a trace buffer, which stores executed instructions and data transfers on the internal buses. The DSU uses a dedicated UART to communicate and may be controlled using an remote debugger[1].

2.2.5 Memory Controller

The memory controller interfaces the processor to the external memory devices. Support is provided for PROMs, SRAMs, SDRAMs and memory mapped I/O devices. The data bus width can be programmed for either 8, 16 or 32-bit memory accesses.

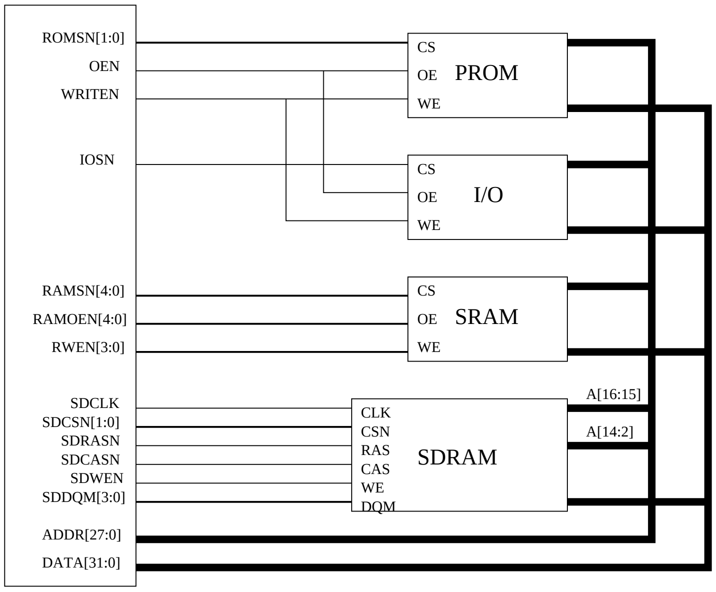

Figure 2.2: Memory Controller Interface Signals

The memory controller is attached to the processor through the AHB and is programmed through three registers (MCR1, MCR2 and MCR3) that govern the setup and operation of the memory controller. It automatically generates all of the control signals required to access the external memory devices and can (optionally) add up to 15 wait-states for slow device access. The memory devices are mapped, by default, according to table 2.1.

2.2.6 Timers

Two 24-bit timers and one 24-bit watchdog are provided on-chip and are clocked by a common 10-bit prescaler. The timers are controlled through the on-chip timer control register. When the watchdog reaches zero, it asserts the WDOG signal, which in turn can be used to generate a system reset.

[1]A DSU monitor is provided by Gaisler Research

2.2.7 Internal UARTs

Two 8-bit UARTs are provided on-chip for serial communication. The UART outputs may be connected to a serial port using an appropriate RS232 standard logic convertor, such as the Maxim MAX3232 [9]. The UARTs are fully functional and are capable of either generating the bit-rate internally, using a 12-bit clock divider, or by using an external source obtained from the parallel inteface. Hardware flow-control, parity checking and stop bit generation are supported.

2.2.8 IRQ Controllers

The interrupt controller manages a total of 15 interrupts, originating from internal and external sources. Each interrupt can be programmed to one of two priority levels. An optional secondary interrupt controller may also be configured and is used to add up to 32 additional interrupts, which can only be used by on-chip peripherals.

2.2.9 Parallel I/O Port

A 32-bit parallel I/O port is provided on-chip. 16-bits are always available and can be individually programmed by software to be an input or an output. An additional 16-bits are only available when the memory bus is configured for 8 or 16-bit operation. Some of the bits have alternate usages, such as UART input/outputs and external interrupt inputs, which are detailed in table 2.3.

I/O port	Function	Type	Description
PIO[15]	TXD1	Output	UART1 Transmit Data
PIO[14]	RXD1	Input	UART1 Receive Data
PIO[13]	RTS1	Output	UART1 Request-to-send
PIO[12]	CTS1	Input	UART1 Clear-to-send
PIO[11]	TXD2	Output	UART2 Transmit Data
PIO[10]	RXD2	Input	UART2 Receive Data
PIO[9]	RTS2	Output	UART2 Request-to-send
PIO[8]	CTS2	Input	UART2 Clear-to-send
PIO[4]	Boot Select	Input	Internal or External Boot Prom
PIO[3]	UART Clock	Input	Use as alternalte UART clock
PIO[1:0]	PROM Width	Input	Defines PROM Width at Boot Time

Table 2.3: Parallel Port Usage (8/16-bit Mode)

2.2.10 PCI Interface

A 32-bit, 33MHz PCI Master/Target Interface is also included as an optional module in the LEON core. It is based on the OpenCores PCI bridge.

2.3 Signalling

Figure 2.3 shows the layout of the top-level LEON entity. The processor has a 32-bit internal address bus, however only 28-bits are visible on the external bus with bits 31-28 being used internally for address decoding.

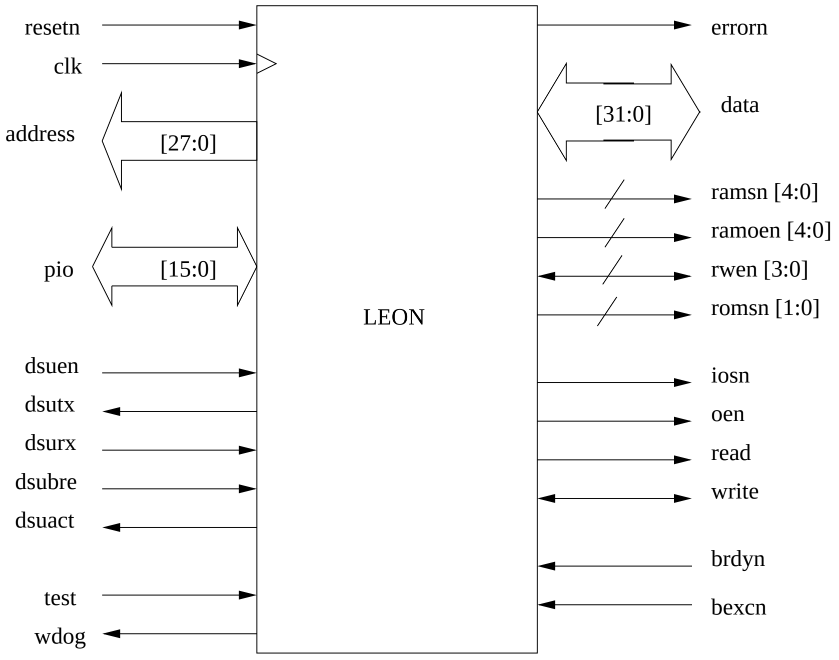

Figure 2.3: Block Diagram of the LEON Microprocessor

The various control signals are summarized in table 2.4. These signals can operate using any signalling standard, which is supported by the target hardware platform. For the purposes of this project, all signals will operate using 3.3V Low Voltage Transistor-to-Transistor Logic (LVTTL).

Signal Type	Label	No.
System Control	resetn, errorn, wdog	3
Address Bus	A27 - A0	28
Data Bus	D7 - D0	8
Parallel Port	pio15 - pio0	16
Bus Control	ramsn[4:0], ramoen[4:0], rwen[3:0], romsn[1:0], iosn, oen, read, write, bdryn, bexcn	22
Debug Support	dsuen, dsurx, dsutx, dsubre, dsuact, test	6
TOTAL:		83

Table 2.4: Signal Classifications of the LEON processor

LEON Memory Accesses

Memory access cycles are controlled by the three memory control registers in LEON. They are used to configure the bus width of both the ROM and RAM chips as well as the amount of wait-states, if any, required for each type of memory access. There are three types of memory access; ROM, RAM and Memory Mapped I/O, however the memory cycle remains similar for each type as shown in figure 2.4.

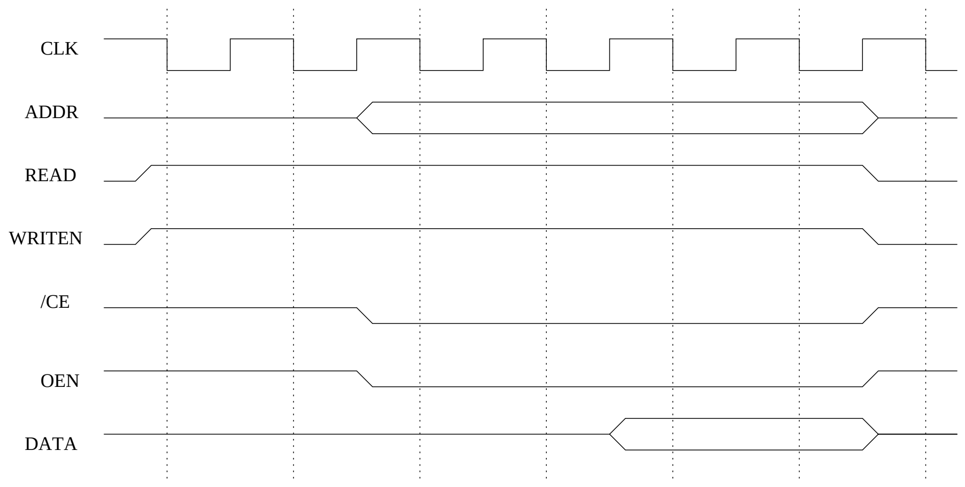

Figure 2.4: Simplified 32-bit LEON Memory Read Cycle

If the processor is configured for 8 or 16-bit data bus widths, the memory cycles will still remain the same apart from having to perform 4 cycles in 8-bit mode and 2 cycles in 16-bit mode in order to retrieve the correct 32-bit quantity from memory.

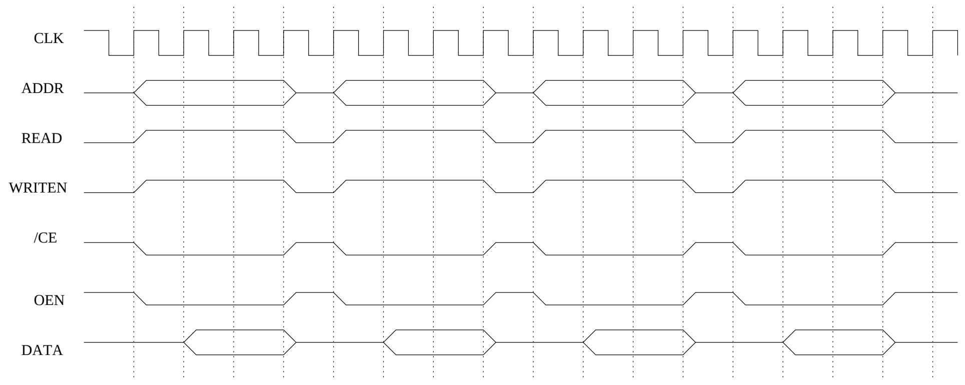

Figure 2.5: Simplified 8-bit LEON Memory Read Cycle

It is also possible to access several consecutive addresses using burst mode access. A burst transfer will be generated when the memory controller is accessed using an AHB burst request. This functionality is not, neccessary for the purposes of the design project.

2.4 Configuration

Configuration of the LEON core is accomplished using records, which are defined in the *target* package. A single VHDL file called `device.vhd` is used to set up the correct configuration record entries, allowing the entire model to be customized for a specific application or target technology. This file may be edited either manually or using a graphical configuration utility based on the linux kernel *tkconfig* scripts.
The model is configured from a master configuration record, which contains a number of sub-records that configure specific modules and functions.

```
type config_type is record
    synthesis  : syn_config_type;     -- synthesis options
    iu         : iu_config_type;      -- integer unit config options
    fpu        : fpu_config_type;     -- floating point unit config options
    cache      : cache_config_type;   -- cache config options
    ahb        : ahb_config_type;     -- ahb config options
    apb        : apb_config_type;     -- apb config options
    mctrl      : mctrl_config_type;   -- memory controller config options
    boot       : boot_config_type;    -- boot config options
    debug      : debug_config_type;   -- debug unit config options
    pci        : pci_config_type;     -- pci config options
    peri       : peri_config_type;    -- peripheral module config options
end record;
```

The synthesis configuration sub-record is used to configure the model for specific synthesis tools and target types. Using this record, technology specific cells within the design can either be automatically inferred or directly instantiated.

```
type targettechs is
  (gen, virtex, virtex2, atc35, atc25,
  atc18, fs90, umc18, tsmc25, proasic, axcel);

-- synthesis configuration
type syn_config_type is record
    targettech  : targettechs;
    infer_ram   : boolean;  -- infer cache and dsu ram automatically
    infer_regf  : boolean;  -- infer the regfile automatically
    infer_rom   : boolean;  -- infer boot prom automatically
    infer_pads  : boolean;  -- infer pads automatically
    infer_mult  : boolean;  -- infer multiplier automatically
    rftype      : integer;  -- regfile implementation option
end record;
```

Any peripheral that is disabled in the configuration record will have its functionality supressed within the model, resulting in a smaller design.

2.5 Simulation

All simulations of the LEON core were performed using Modelsim-5.7SE[2]. Generic testbenches are provided with LEON, which allow simulation of the model with 32, 16 or 8-bit data buses using precompiled test programs. All simulations were performed using the 8-bit bus configuration as this is the configuration which would be used in the actual design project.
The three main classes of testbenches provided with the model are:

1. *Functional Tests:* These test most on-chip functionality using either 8, 16 or 32-bit external static RAM or else 32-bit external SDRAM.

2. *Memory Tests:* These test on-chip memory with patterns of 0x55 and 0xAA, again using either 8, 16 or 32-bit data bus widths.

3. *Full Tests:* These provide full functional and memory tests.

Several simulations of LEON were performed before any alterations were made to the model. The purpose of this was to gain familiarity with the Modelsim environment as well as the operation of the processor. The following output is generated by the simulator when running the standard *func_8* testbench without any modifications.

```
## *** Starting LEON system test ***
## Memory interface test
## Cache test
## Register file
## Interrupt controller
## Timers, watchdog and power-down
## Parallel I/O Port
## UARTs
## Test completed OK, halting with failure
# ** Failure: TEST COMPLETED OK, ending with FAILURE
```

All of the testbenches operate by simulating ROM and RAM devices, which can be initialised with data from special files, and interfacing them with the processor model in the same way that would be done in a real hardware system. Due to this layout, arbitrary programs suitable for use with the LEON architecture, may be run from within the testbenches, adding to their versatility.
The simulations allow all of the internal signalling within the processor to be viewed, aiding in the process of debugging new code or verifying that any changes that have been made operate in the fashion that was intended.
Figure 2.6 shows as sample of the output waveforms that are obtained from simulations performed on the model using the 8-bit functional testbench.

[2]Available from http://www.model.com

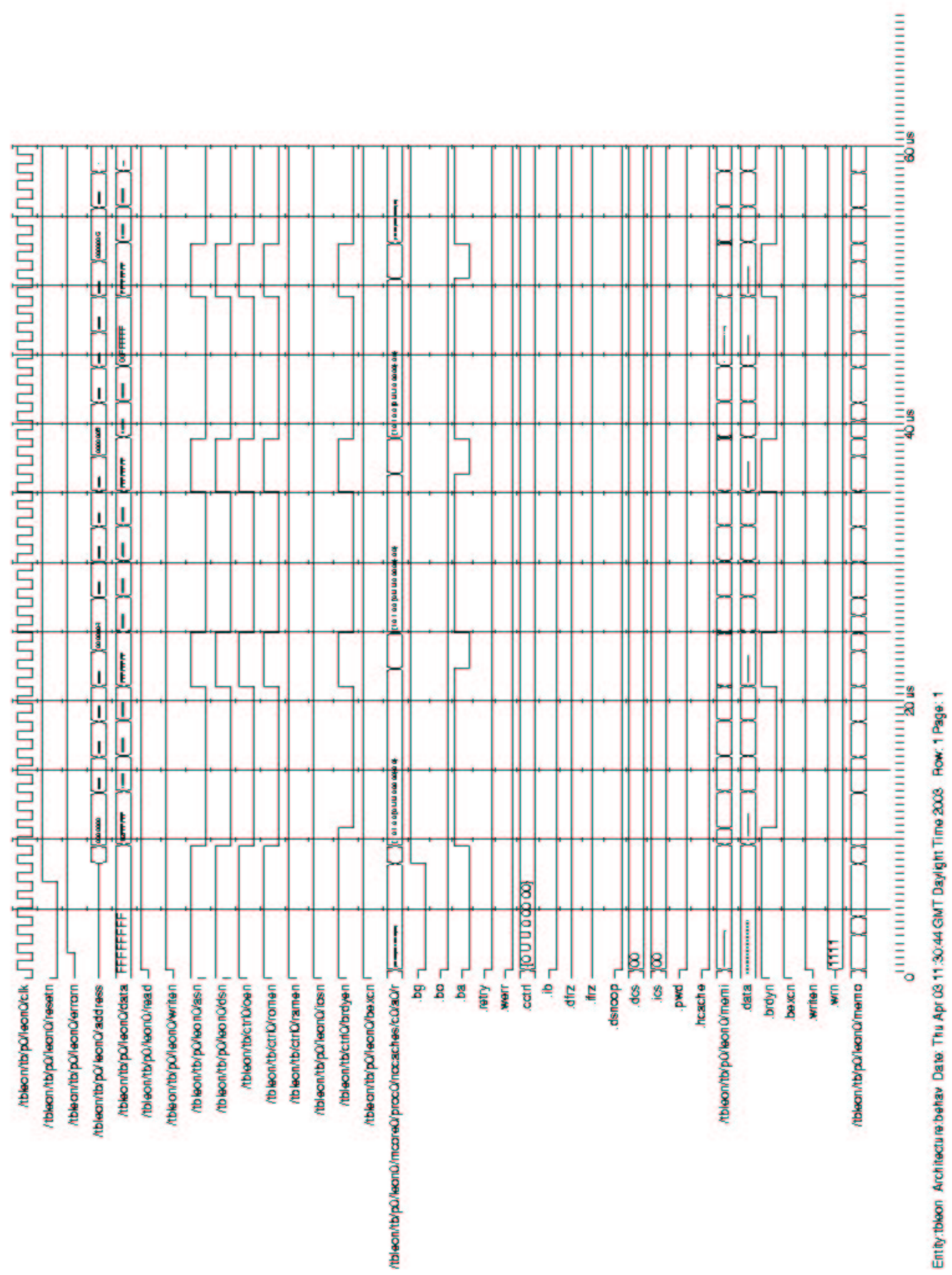

Figure 2.6: Waveform results of standard tb_func8 testbench

2.6 Synthesis

The Xilinx Synthesis Technology (XST) suite of HDL compilation tools was used to build and synthesise the LEON core. The following commands were used to achieve a final bit-file, targetted at the VirtexII platform.

```
xst -ifn leon.xst
ngdbuild leon -uc leon-proj.ucf
map -detail leon.ngd
par leon.ncd leon_par.ncd
bitgen leon_par.ncd leon.bit
```

The `xst` program was used to compile and synthesise all of the relevant files making up the LEON model. These files were listed in the correct compilation order in the "*leon.xst*" file, which ensured that no dependency issues arose during compilation if the source files were analysed in the wrong order.
The `ngdbuild` program was used to create a single "ngd" design file from the various synthesised project files. This included options from the user constraints file ("*leon-proj.ucf*") telling the program which external IO Pads to lock the signals to.
The `map` program was used to map the design from the "ngd" file and create a "ncd" file specific to the target FPGA platform, which in this case was the VirtexII XC2V1000[3].
The `par` program was used to place and route all of the signals and nets within the design. This is the final step in compilation of the design before a bit-file is created.
Finally, the `bitgen` program was used to create the final bit-file containing the complete design. This file is suitable for download directly into the FPGA if required, however for the purposes of the project, this file is then formatted by the PROM formatter utility before being downloaded into the configuration PROM, which in turn programmes the FPGA automatically on power-up of the prototyping board.
At this point no modifications had been made to the LEON core so the default synthesis options and constraints were used to test the synthesis process.

Chapter 3

Modifying the LEON Core

While evaluating the LEON core, it became clear that several major modifications would have to be made in order for it to be suitable as a replacement for the MC68008. It would have to be as simple as possible and include only the on-chip components essential to the operation of the processor.
These changes would drastically reduce the performance of the processor by an estimated 95% [1], however this was not an important issue for the purposes of the design project. This chapter describes in detail, all of the changes made to the LEON core and the reasons that they were made. Throughout the project, emphasis was placed on maintaining the configurability of the model, so special care was made to make sure that the effects of any changes made to the design could be reversed through a configuration option, which could be easily made, using the graphical configuration utility that had originally been supplied with the model. This meant that any combination of changes could be implemented while being able to recover the original operation of the processor at any time.

3.1 Identifying Project Tasks

The major tasks that would have to be achieved in order to make LEON suitable as a replacement for the MC68008 were first identified. If all of these changes could be implemented successfully, then the LEON processor would be deemed suitable for use as the basis of implementing an upgraded version of the microprocessor design project, while keeping to a similar format in terms of build time and complexity.

Configurable Modules

Due to the high level of configurability of the LEON model, it was already possible to disable some of the internal modules which had been identified as uneccessary for the purposes of the project. These included the debug support unit, the SDRAM controller and the secondary IRQ controller. It was also already possible to configure the data bus for 8-bit operation, which was a major advantage as it meant that the LEON processor

[1]87.5% reduction due to the lowered clock speed and 25% reduction due to the removal of the caches

could be configured to operate with the same bus width as was present in the MC68008. It was important to be able to suppress as many of the uneccessary peripherals as possible in order to simplify the core. Any peripheral that was not required for the operation of the processor and could not be removed, would have to be altered in such a way that its functionality was suppressed.
It would be necessary to be able to remove the caches and to alter the memory map and controller in a suitable manner if the LEON processor was to be used in the new design project. Failure to implement these modifications successfully would render the processor unsuitable for use as a replacement for the MC68008.

Clock Speed

The first task identified would be to see what effect, if any, that a reduced clock speed would have on the processor. It was important to reduce the clock speed as the external components of the microprocessor system would be wire-wrapped together and a fast system clock speed might make the system unstable due to poor connections between the component pins and the connecting wire.

Caches

The second task would be to remove the instruction and data caches. This was an important task as the only method that the students would have to monitor the operation of the processor was by capturing and examining the activity on the address and data buses. If the caches were present within the processor, some of the memory requests made could be hidden from the external buses if a cache hit occurred. The caches would also add an extra level of complexity to the processor, which was not desirable.

Internal UARTs

The third task would be to remove the internal UARTs. This was due to the fact that one of the tasks the students would have to undertake when building the microprocessor system would be to implement UARTs using external circuitry and although retaining the internal UARTs would not hinder this process, they would add unecessary logic and complexity to the core.

Reset Generator

The fourth task would be to suppress the functionality of the reset generation unit. This was due to the fact that the reset generation circuitry would have to be implemented externally by the students and so was not required within the model.

Memory Map

The fifth task would be to remove the internal memory map for the processor. Its functionality would be replaced by external logic as part of the design project.

Memory Controller

Finally, any bus transaction signalling that was generated within the processor would have to be disabled. A new method of performing bus transactions would then have to be devised and implemented. This new system would have to be easily understood by the students.

3.2 VHDL Model Heirarchy

Figure 3.1 details the design layout of the LEON core, showing the individual VHDL functional modules and their positions within the overall heirarchy of the design. Signals between the modules are passed in records and each module is individually configurable from within the *device.vhd* configuration file.

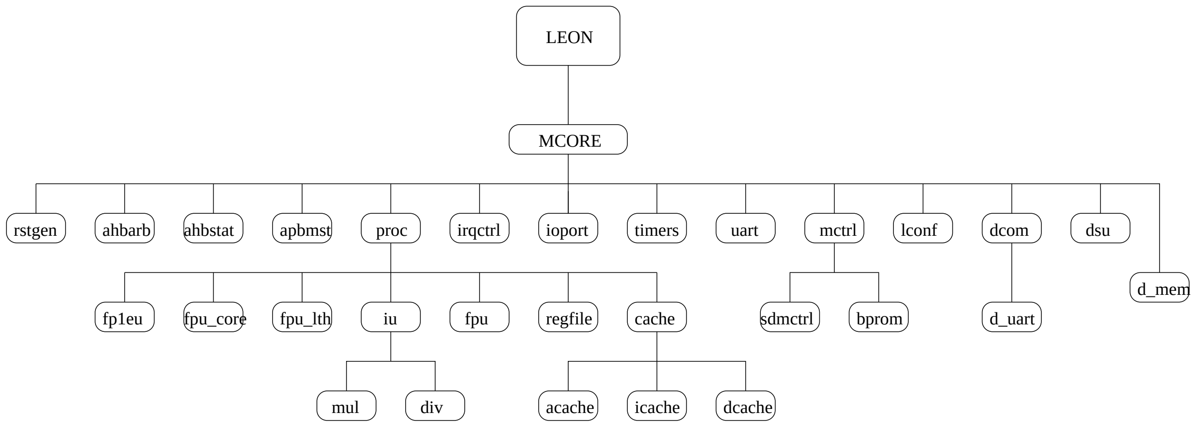

Figure 3.1: Layout of the LEON Core

3.3 Clock Speed

The first step taken in modifying the core was to simulate the behavioural model operating at a clock frequency of 6.25 MHz. This was important as the model had initially been designed to operate at 50 MHz. However, it was felt that the system clock frequency should be reduced as much as possible due to the fact that the external components would be wire-wrapped together and a low clock speed would reduce the chance of signalling errors due to faulty connections. 6.25 MHz was chosen as the operating clock frequency as it was easily obtainable from the original 50 MHz signal and was approximately the same frequency as used by the MC68008 in the original project. Tests were first run using the generic 8-bit data bus testbench (provided with the model) running at 50 MHz, as a reference.
The testbench and model were then configured to run at approximately 6.25 MHz and simulations re-run. This was done by changing the *clkperiod* value in the testbench from 20 to 160, resulting in a system frequency of 6.25 MHz

```
clkperiod : integer := 20; -- 50MHz
```

3.3.1 Simulation

The test program completed successfully for both instances and comparison of the waveforms for each run showed no problems with the operation of the simulation. This result implied that there would be no problem in reducing the clock speed of the processor once it was running on the FPGA, however to only way to test this fully would be once the model had been fully synthesised.

3.4 Caches

The Integer Unit (IU) is connect to the AHB bus through the instruction and data caches. The complete removal of the caches would have meant a major re-write of the IU so it was decided to leave the main functionality of the caches intact. The caches were instead altered so that no memory accesses would be cached and that cache misses would be continually forced. This would have the effect of totally supressing the cache functionality within the model. The internal cacheability table was also removed and all address space locations defined as *n*on-cacheable by default. Finally, the on-chip cache memory was removed as this was now redundant.

3.4.1 Alterations to the Configuration Records

The option to disable the instruction and data caches was entered into the graphical configuration utility, however this new functionality still had to be supported within the model. This was done by adding a boolean variable to control whether the caches had been enabled or disabled, to the cache configuration record as defined in *target.vhd* file.

```
type cache_config_type is record
    enable      : boolean;    -- enable/disable cacheing
    isets       : integer range 1 to MAXSETS;    -- no of sets in icache
    isetsize    : integer;    -- icache size per set in kB's
    ilineisze   : integer;    -- no of words per icache line
    ireplace    : cache_replace_type;    -- icache replacement algorithm
    ilock       : integer;    -- icache locking
    dsets       : integer range 1 to MAXSETS;    -- no of sets in dcache
    dsetsize    : integer;    -- dcache size per set in kB's
    dlinesize   : integer;    -- no of words per dcache line
    dreplace    : cache_replace_type;    -- dcache replacement algorithm
    dlock       : integer;    -- dcache locking
    dsnoop      : dsnoop_type;  -- dcache snooping
    drfast      : boolean;      -- dcache fast read-data gen
    dwfast      : boolean;      -- dcache fast write-data gen
end record;
```

The *config.vhd* file was then altered to include the new CACHE_ENABLE variable within the model.

```
constant CACHE_ENABLE : boolean := cache_config.enable;
```

At this point, the source code could be altered to include the new changes.

3.4.2 Alterations to the Source Code

Removal of the Cacheability Table

The first task was to remove the internal mapping designating areas which were cacheable and not cacheable. All areas within the address space were designated as non-cacheable instead.

```
function is_cacheable(haddr : std_logic_vector(31 downto 24))
  return std_logic is variable hcache : std_logic;
begin
    if CACHE_ENABLE then
        if(haddr(31) = '0') and (haddr(30 downto 29) /- "01") then
            hcache := '1';
        else
            hcache := '0';
        end if;
    else
        hcache := '0';
    end if;
return(hcache);
end;
```

Cache Control Register

Both the instruction and data caches are controlled using a cache control register. This register controls the functionality and state of both of the caches and can be used to disable them.

```
if CACHE_ENABLE then
    if(r.cctrl.ifrz and iuo.intrack and r.cctrl.ics(0)) = '1' then
        v.cctrl.ics = "01";
    end if;
    if(r.cctrl.dfrz and iuo.intrack and r.cctrl.dcs(0)) = '1' then
        v.cctrl.dcs = "01";
    end if;
else
    v.cctrl.ics = "00";    -- disable instruction cache
    v.cctrl.dcs = "00";    -- disable data cache
end if;
```

By setting the *ics* and *dcs* portions of the control register to “00” in this way, the instruction and data caches are not allowed to leave the disabled state.

Cache Memory

The cache memory was then removed as it was no longer needed. This also had the effect of reducing the size of the overall design.

3.4.3 Simulation

Simulations were run using an infinite loop program. This test proved that the caches were passing the correct information to the IU while not storing the information.

3.5 Internal UARTs

The UARTs are seperate entities, which are generated within the *mcore* module. In order to remove them from the model, their generation had to be disabled and any references to them removed. This was done by adding a `boolean` variable for each UART, within the configuration records that control the generation of the UARTs. The generation of the UART modules could then be indivually controlled through the graphical configuration interface.

3.5.1 Alterations to the Configuration Records

Once the UART enable variables had been added into the graphical configuration interface, their functionality had to be supported within the LEON model. The first step was to place the entries into the correct configuration record. The `peri_config` record was chosen for this purpose. The *uart1_en* and *uart2_en* options to enable or disable the UARTs individually were added at the end of the appropriate record in the *target.vhd* file.

```
type peri_config_type is record
    cfgreg      : boolean;    -- LEON config register enable
    ahbstat     : boolean;    -- AHB status register enable
    wprot       : boolean;    -- RAM write-protection enable
    wdog        : boolean;    -- watchdog enable
    irq2en      : boolean;    -- second interrupt controller enable
    ahbram      : boolean;    -- AHB RAM enable
    ahbrambits  : integer;    -- Address bits in AHB RAM
    uart1_en    : boolean;    -- First UART enable
    uart2_en    : boolean;    -- Second UART enable
end record;
```

The corresponding entries were then placed into the *config.vhd* file in order to make the added variable visible within the entire model.

```
constant UART1_EN : boolean := peri_config.uart1_en;
constant UART2_EN : boolean := peri_config.uart2_en;
```

At this point, the source code could be altered to include the changes that these new variables would impose.

3.5.2 Alterations to the Source Code

The two UARTs were generate separately within the *mcore* module. "`if`" statements were used to disable the code that generated them.

```
uart1on : if UART1_EN generate
    uart1i.rxd    <= pioo.rxd(0);
    uart1i.ctsn   <= pioo.ctsn(0);
    uart1i.scaler <= pioo.io8lsb;

    uart1 : uart port map(
        rst => rst, clk => clk, apbi => apbi(6), apbo => apbo(6),
        uarti => uart1i, uarto => uart1o );
end generate;

uart2on : if UART2_EN generate
    uart2i.rxd    <= pioo.rxd(1);
    uart2i.ctsn   <= pioo.ctsn(1);
    uart2i.scaler <= pioo.io8lsb;

    uart2 : uart port map(
        rst => rst, clk => clk, apbi => apbi(7), apbo => apbo(7),
        uarti => uart2i, uarto => uart2o );
end generate;
```

Once the generation of the UARTs had been disabled, any reference to their functionality also had to be removed from the model. The two UARTs were assigned IRQs 3 and 2 respectively. Both of these interrupt levels had to be unassigned when the UARTs were not enabled within the model. This was done by adding a "`when`" statement to the IRQ assignments within the interrupt controller.

```
    irqi.irq(3) <= uart1o.irq when UART1_EN else '0'; -- First UART
    irqi.irq(2) <= uart2o.irq when UART2_EN else '0'; -- Second UART
```

Finally, the on-chip registers for the two seperate UARTs had to be removed from the memory map. These were defined in the *apbmst.vhd* file.

```
-- UART1    0x70 to 0x7C
when "00011100" | "00011101" | "00011110" | "00011111" =>
    if UART1_EN then esel := '1'; bindex := '6'; end if;
-- UART2    0x80 to 0x8C
when "00100000" | "00100001" | "00100010" | "00100011" =>
    if UART2_EN then esel := '1'; bindex := '7'; end if;
```

3.5.3 Simulation

Simulations were then performed to see what effect these changes had on the model. The standard 8-bit functional tests were run. The tests failed the UARTs proving that they were no longer functional.

3.6 Reset Generation

The reset generation module was used to extend the active-low reset signal within the processor on detection of an external reset event. This was necessary to ensure that all of the signals within the processor were given time to stabilise during reset. As one of the tasks the students would be required to do when building the microprocessor system would be to implement the reset circuitry externally, it was necessary to remove the functionality of the reset generator from the model.

3.6.1 Alterations to the Configuration Records

The peripheral configuration record was altered to include the `reseten` option to enable or disable the reset generator module.

```
type peri_config_type is record
    cfgreg        : boolean;    -- LEON config register enable
    ahbstat       : boolean;    -- AHB status register enable
    wprot         : boolean;    -- RAM write-protection enable
    wdog          : boolean;    -- watchdog enable
    irq2en        : boolean;    -- second interrupt controller enable
    ahbram        : boolean;    -- AHB RAM enable
    ahbrambits    : integer;    -- address bits in AHB RAM
    reseten       : boolean;    -- reset generation enable
    uart1_en      : boolean;    -- first UART enable
    uart2_en      : boolean;    -- second UART enable
end record;
```

The *config.vhd* file was then altered to include the new reset option.

```
constant RESETEN : boolean := peri_config.reseten;    -- reset gen enable
```

3.6.2 Alterations to the Source Code

The generation of the reset delay could then be removed from the model. This was done by bypassing the shift register that created the delay and passing the signal straight through the module into the rest of the processor.

```
if RESETEN then
    rsttmp <= r(4) and r(3) and r(2);
else
    rsttmp <= rstin;
end if;
```

This meant that the reset signal inside the processor would be held low for the same amount of time as the external reset signal.

3.6.3 Simulation

In order to establish that the reset circuitry had actually been disabled, simulations were run on the model. It was verified that the reset signal stayed low for only one clock cycle after the reset button had been pressed instead of the usual five clock cycles. This proved that the internal reset circuitry was disabled.

3.7 Memory Map and Controller

Memory Map

At this stage, the on-board memory map for the LEON core had to be removed. It was decided, however, not to remove the address mappings for the on-chip registers or the debug support unit. The only changes made were to remove all of the ROM, RAM and I/O mappings from the source code.
Due to the fact that only 28-bits of the 32-bit address bus would be visible externally, an address space of only 160MB would be available for the students to utilize. This would not be a problem though, as a maximum of 1MB would only ever be needed to implement the design project. The on-chip and debug support registers would still be fully accessible as their internal mappings had not been altered.

Address Range	Size	Mapping	Module
0x00000000 - 0x09FFFFFF	160 MB	Mappable Area	External Circuitry
0x10000000 - 0x7FFFFFFF	1792 MB	UNUSED	N/A
0x80000000 - 0x8FFFFFFF	256 MB	On-chip Registers	APB Bridge
0x90000000 - 0x9FFFFFFF	256 MB	Debug Support Unit	DSU

Table 3.1: Project Memory Map

Memory Controller

The memory controller had to be altered in order to suppress any of the bus transaction signals that were generated internal to the model. These inlcluded the `ramoen[3:0]`, `ramsn[3:0]`, `romsn[1:0]` and `rwen[3:0]` signals that were responsible for controlling memory accesses to peripheral devices.
A new bus transaction protocol would have to be implemented in place of the signals that had been disabled. It was decided to implement a new system similar to that of the Motorola MC68008 processor. The `BRDYEN` signal would be altered to operate in a similar fashion to the MC68008 `/DTACK` signal. Address and data strobe signals would then be added to the LEON processor to make it compatible with the Motorola signals.

3.7.1 Alterations to the Configuration Records

The first step involved in implementing these changes was to add a variable to the configuration records, which could control whether or not the internal address maps and signalling should be enabled or disabled. This variable would be called *MEMSEL*.

```
-- memory controller configuration
type mctrl_config_type is record
    memsel    : boolean;    -- enable chip select signals
    bus8en    : boolean;    -- enable 8-bit bus operation
    bus16en   : boolean;    -- enable 16-bit bus operation
    wendfb    : boolean;    -- enable wen feedback to data bus drivers
    ramsel5   : boolean;    -- enable 5th ram select
    sdramen   : boolean;    -- enable sdram controller
    sdinvclk  : boolean;    -- invert sdram clock
end record;
```

The corresponding variable entry was then placed into the *config.vhd* file to make the new configuration option visible within the model.

```
constant MEMSEL : boolean := mctrl_config.memsel;
```

The source code for the memory controller could now be altered to include any changes that this new variable would have.

3.7.2 Alterations to the Source Code

Changing the Memory Map

The first step taken was to remove the part of the memory map that dealt with access types. The default access type was set to `ram`, regardless of the address being accessed.

```
    if MEMSEL then
        case haddr(30 downto 28) is
            when "000" | "001" => area := rom;    -- ROM address space
            when "010" | "011" => area := io;     -- IO address space
            when others => area := ram;           -- RAM address space
        end case;
    else
        area := ram;    -- Assume RAM for all address space
    end if;
```

Disabling the Bus Signals

The next step was to disable the appropriate bus signals. The actual generation of these signals was left unaltered, however their register vaulues were all tied high, preventing their state from changing within the model. This effectivly disabled all of the bus signals within the model.

```
if MEMSEL then
    memo.ramsn(4 downto 0)  <= r.ramsn;
    memo.ramoen(4 downto 0) <= r.ramoen;
    memo.romsn              <= r.romsn;
    memo.oen                <= r.oen;
    memo.iosn               <= r.iosn(0);
else
    memo.ramsn(4 downto 0)  <= "11111";
    memo.ramoen(4 downto 0) <= "11111";
    memo.romsn              <= "11";
    memo.oen                <= '1';
    memo.iosn               <= '1';
end if;
```

Address and Data Strobes

Once the signals had been disabled, address and data strobe signals, with a similar function to their MC68008 counterparts, had to be generated within the model. The fact that the bus transaction signals were still being generated internally within the model was taken advantage of when generating the address and data strobe signals.

```
-- Address and Data strobe generation
if MEMSEL then
    as <= '1';
    ds <= '1';
else
    if((r.ramsn = "11111") and (r.romsn = "11") and (r.iosn = "11")) then
        as <= '1'; else as <= '0'; end if;

    if((r.ramoen = "11111") and (r.oen = '1')) then
        ds <= '1'; else ds <= '0'; end if;
end if;
```

The address strobe is activated if any of the ROM, RAM or IO selects are active. Similarly, the data strobe is activated if any of the output enable signals are active. If, however, the core is configured to generate the signals internally, the address and data strobes are held inactive.

Data Transfer Acknowledge

The next step in altering the memory controller was to provide a method whereby the processor could be notified, by external means, to end the current memory access cycle. In the MC68008, a signal called `/DTACK` was provided for this purpose. The unmodified LEON core provided for a signal called `BRDYN`, which could be used in a similar way, to extend the number of processor wait-states during an I/O access.

The operation of this signal was modified so that it could be used during any type of memory access instead of just I/O accesses, as long as the processor was operating in 8-bit mode. The state machine controlling 8-bit data bus accesses was altered so that would add continuous wait-states until it received the active-low `BRDYN` signal. Once the signal had been received, the processor would latch the data on the bus and proceed with the next memory access, as required.

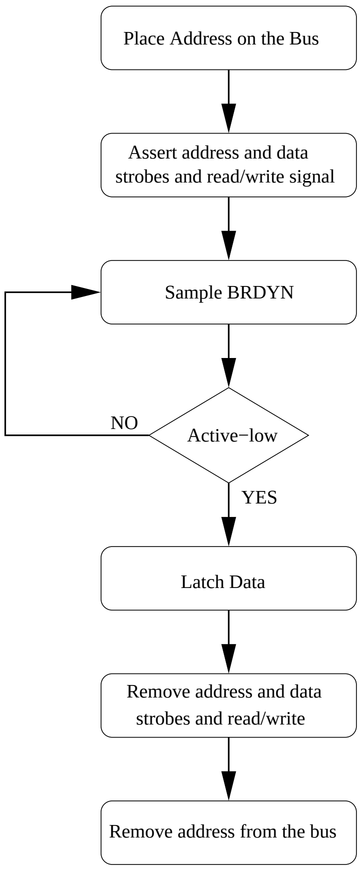
Place Address on the Bus
Assert address and data strobes and read/write signal
Sample BRDYN
NO
Active−low
YES
Latch Data
Remove address and data strobes and read/write
Remove address from the bus

Figure 3.2: State flow diagram for updated LEON memory cycle

Memory Configuration Registers

The final step in altering the memory controller was to initialise the three memory configuration registers with the correct data in order to make the controller suitable for operation with the format required by the design project. Only the first and second memory configuration registers needed to be initialised as the third is exclusively used to control the SDRAM interface, which will not be used in the design project.

```
-- memory configuration register 1
v.mcfg1.romrws := "0000";     -- no rom read wait states
v.mcfg1.romwws := "0000";     -- no rom write wait states
v.mcfg1.romwidth := "00";     -- 8-bit rom data bus
```

```
v.mcfg1.iows := "0000";        -- no I/O wait states
v.mcfg1.bexcen := '1';         -- enable bus exception signalling
v.mcfg1.brdyen := '1';         -- enable BRDYN for ROM and I/O accesses
v.mcfg1.iowidth := "00";       -- 8-bit I/O data bus

-- memory configuration register 2
v.mcfg2.ramrws := "0000";      -- no ram read wait states
v.mcfg2.ramwws := "0000";      -- no ram write wait states
v.mcfg2.ramwidth := "00";      -- 8-bit ram data bus
v.mcfg2.rambanksz := "0111";   -- 128kB per RAM bank
v.mcfg2.rmw := '0';            -- read-modify-write
v.mcfg2.brdyen := '1';         -- enable BRDYN for RAM accesses
```

These default values can be over-written at any stage during program execution by writing to the appropriate on-chip register. The advantage of having them automatically set up in hardware is that the registers do not have to be set up by software routines before the processor may be used.

3.7.3 Simulation

At this point, testing with the standard test benches could not be accomplished as too much functionality had been removed from the model. To get around this, a new test bench (called "*tb_projnew*") was created, which emulated the bus transaction signals for the processor. This testbench would be used temporarily until the external control logic had been implemented.

3.8 Model Synthesis

Once all of the modifications had been made, the LEON model had to be synthesised and compiled into a bit-file, which could be downloaded onto the target hardware. This process involved configuring the model with the appropriate options, synthesising it, applying the user constraints to lock the processor signals to the correct FGPA pins, placing and routing the netlist and finally creating the bit-file before downloading the LEON processor to the FPGA. The synthesis followed the same pattern as outlined in section 2.6.

3.8.1 Configuration

The first step in creating a bit-file was to set the correct configuration options for the model using the graphical configuration utility provided.
All of the changes made to the model could easily be activated or deactivated using this utility. The actual final configuration file used for the model, including all of the new configuration options, can be found appendix B.1.

3.8.2 Synthesis

Once the model had been configured correctly, the source code was compiled using the Xililnx `xst` programme. A summary of the synthesis report is given below.

```
-- Target Parameters
Target Device         : xc2v1000-fg256-5
Output File Name      : leon
Target Technology     : virtex2
Speed Grade           : 5

--HDL Synthesis Report
Macro Statistics:
# FSMs                : 6
# ROMs                : 3
# Registers           : 349
# Counters            : 2
# Multiplexers        : 478
# Tristates           : 48
# Adders/Subtractors  : 25
# Comparators         : 11
# XORs                : 16

-- Final Results
Macro Statistics:
# ROMs                : 3
# Registers           : 361
# Multiplexers        : 30
# Tristates           : 48
# Adders/Subtractors  : 16
# Comparators         : 10

Design Statistics:
# IOs                 : 110

Cell Usage:
# BELS                : 6207
# Flipflops/Latches   : 1594
# RAMs                : 2
# Clock Buffers       : 1
# IO Buffers          : 105

-- Timing Summary
Speed Grade           : -5
    Minimum Period    : 17.096 nS (Maximum Frequency  : 58.493 MHz)
    Minimum Input Arrival Time Before Clock     : 2.691 nS
    Maximum Output Required Time After Clock    : 10.849 nS
    Maximum Combinational Path Delay            : No Path found
```

```
---------------------------------------
CPU       : 1872.48 Secs / 1894.20 Secs
Elapsed   : 1872.00 Secs / 1894.00 Secs
```

3.8.3 Constraints

The user constraints were then applied to the model using the `ngdbuild` tool, locking the signals to the appropriate output pins on the Xilinx chip. A table of the constraints used for the project can be found in table B.1 in the appendix. These constraints were created for the LEON model when running on the VirtexII FPGA using the Xilinx prototyping board. Though most of the pin allocations were arbitrary, an attempt was made to place signals close to each other according to functionality.

3.8.4 Mapping, Placing and Routing

After the user constraints had been incorporated into the design, it was mapped into a target specific netlist. The following report was generated by the mapping tool, showing that approximately 62% of the available logic inside the VirtexII chip was being utilized by the LEON core (including the control logic block).

```
-- Design Information
    Command Line        : map -detail leon.ngd
    Target Device       : xc2v1000
    Target Package      : fg256
    Target Speed        : -5

-- Design Summary
    Number of Errors            : 0
    Number of Warnings          : 1
    Number of Slices            : 3195/5120  (62%)
    Number of Slice FlipFlops   : 1593/10240 (15%)
    Total Number 4 Input LUTs   : 5374/10240 (52%)
    Number of Bonded IOBs       : 106/172    (61%)
    IOB Flipflops               : 1
    Number of Block RAMs        : 2/40       (5%)
    Number of GCLK              : 1/16       (6%)
Total equivalent gate count for design    :  178,222
Additional JTAG gate count for IOBs       :  5,088
```

The model was then placed and routed using the `par` programme, which resolved any unrouted signals within the design. This was the last compilation step in generating the LEON core.

3.8.5 Bitfile Generation

The final stage was to create the final bit file that would be programmed onto the VirtexII FPGA. This was done using the `bitgen` tool and did not produce any errors or warnings.

3.9 Downloading the Core

The LEON bit-file was then formatted for the XC18V04 chip using the PROM formatter utility that came with the Xilinx Software. Once complete, it was downloaded onto the prototyping board using the Xilinx iMPACT programme. This meant that the FGPA would automatically be programmed with the LEON bit-file when power was applied to the prototyping board, eliminating the need to keep reprogramming the FPGA before every use. The bit-file was downloaded in the same way as described in section 1.5.

Chapter 4

Hardware Design and Assembly

Once all of the modifications discussed in chapter 3 had been made to the LEON core, the process of designing and building the hardware aspect of the microprocessor system began. This chapter describes the steps taken in implementing the prototype hardware design for the project and details the decisions made during the design process.
The first step in designing the microprocessor hardware is to re-implement any of the control logic that had been removed from the LEON model, such as the bus transaction signalling and memory map.

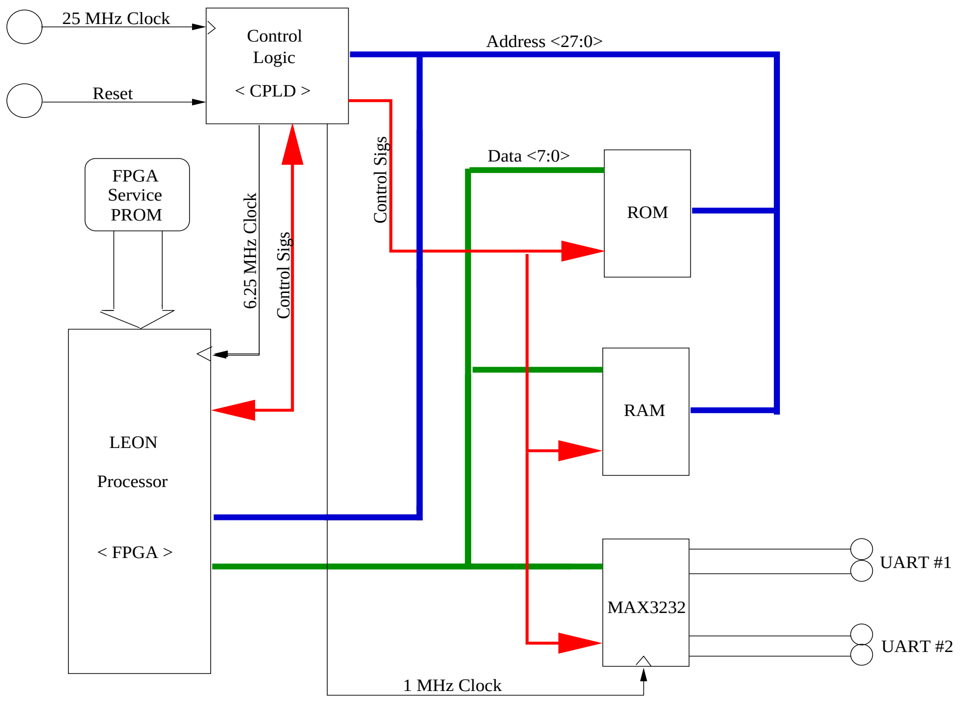

Figure 4.1: Block Diagram of the Proposed Project Hardware

The basic layout of the hardware required in the project is shown in figure 4.1. The system comprises a ROM chip, a RAM chip, a voltage convertor for the RS232 interface and the control logic block.

4.1 Control Logic

All of the control logic was written in VHDL and implemented as a seperate entity within the same FPGA as the LEON core. This was done for the purposes of testing the operation of the hardware. The students undertaking this project will be required to implement the control logic in a separate FPGA or CPLD to the LEON core.
The main advantages of this is that they are not required to have any knowledge about the implementation of the LEON core itself and that all of the control logic may be written in a Hardware Description Language (HDL) such as VHDL or Verilog and implemented entirely within one chip, moving away from the need to implement external circuitry or GAL logic and cutting down on the number of ICs required in the overall design. This means that the students have more time to concentrate on the logic required to get the processor to operate correctly and have to spend less time building the project circuitry.
A brief description of each of the VHDL modules within the control logic block and their operation follows. All of the source code is made available under the GNU-LGPL, in keeping with the spirit of LEON.

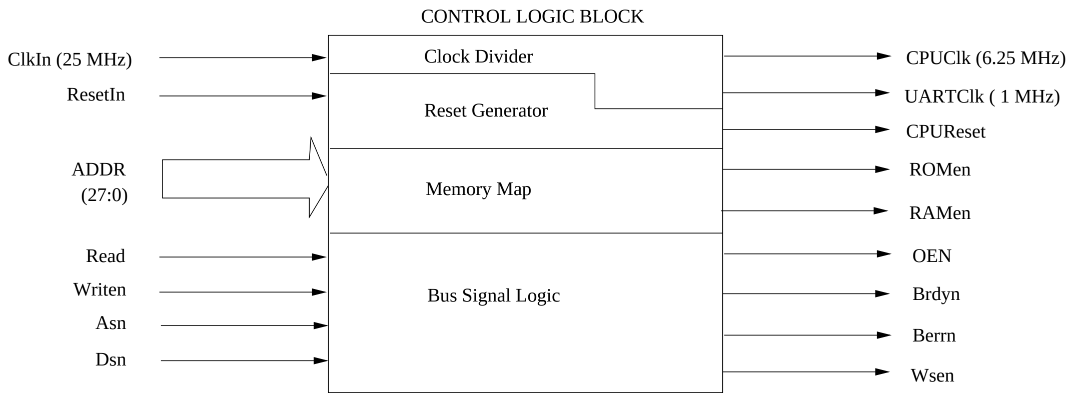

Figure 4.2: Block Diagram of the Control Logic

Figure 4.2 shows a block diagram of all of the control logic signals that are required to be implemented in order to make the processor operational. The signals are generated in three seperate VHDL modules, which are outlined below.

1. The Clock Divider Module
2. The Reset Signal Generator Module
3. The Memory Controller Module

These three modules are then combined together into one control entity. In order to demonstrate the operation of the system within the time constraints of the project, the control entity was incorporated into the LEON core as a separate module, with the appropriate signalling being "wired" together using VHDL signal assignments.

4.1.1 Clock Divider

The function of the clock divider is to obtain a 6.25MHz system clock signal and a 1MHz UART reference signal from the 25MHz on-board signal. This is done by implementing a 5-bit counter and using bits 2 and 4 to recover the system and UART clock signals.

```
entity clkdiv is
  port(
    clkin       : in std_logic;     -- 25MHz reference signal
    clkout_cpu  : out std_logic;    -- (25/4) MHz CPU system clock
    clkout_uart : out std_logic     -- (25/16) MHz UART signal
  );
end clkdiv;
```

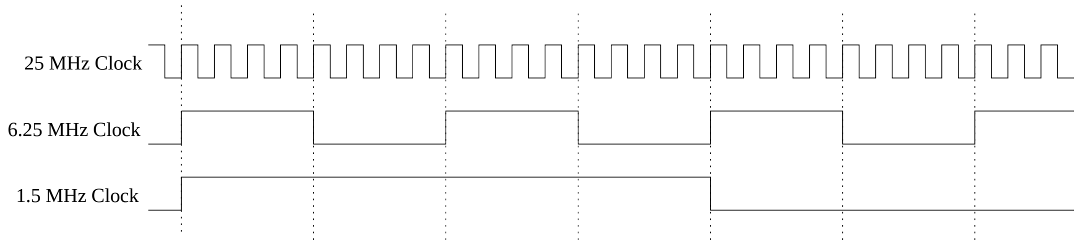

Figure 4.3: Diagram of the clock divider output

The second bit of the counter is used to obtain the 6.25MHz clock signal as it effectively divides the input signal by 4 (2^2). The fifth bit is used to obtain the 1.5MHz clock signal as it effectively divides the input signal by 16 (2^4).

4.1.2 Reset Generator

The purpose of the reset generator is to ensure an active-low reset signal for a minimum period that allows all of the on-board devices, as well as the processor itself, to reset correctly.

```
entity rstsig is
  port(
    clk     : in std_logic;     -- CPU system clock signal
    rstin   : in std_logic;     -- reset button signal (active low)
    rstout  : out std_logic     -- delayed system reset signal
  );
end rstsig;
```

This was implemented using a 4-bit shift register, which would extend the reset signal by a further 4 clock cycles after the reset button had been pressed, as shown in figure 4.4. This delay provides adequate time for all of the system devices to be reset.

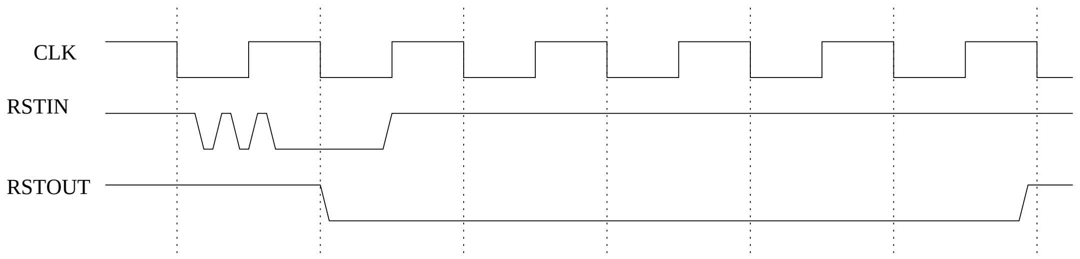

Figure 4.4: Diagram of Reset Signal Generation

4.1.3 Memory Controller

The memory controller module is responsible for implementing the desired project memory-map. It needs to provide chip select signalling for both the ROM and RAM chips as well as the various other bus control signals necessary.

```
entity memctrl is
  port(
    rst     : in std_logic;    -- system reset
    clk     : in std_logic;    -- system clock (6.25MHz)
    addr    : in std_logic_vector(27 downto 0);    -- address bus
    read    : in std_logic;    -- read cycle
    writen  : in std_logic;    -- write cycle
    asn     : in std_logic;    -- address strobe
    dsn     : in std_logic;    -- data strobe
    rom0en  : out std_logic;   -- ROM #0 select
    ram0en  : out std_logic;   -- RAM #0 select
    outen   : out std_logic;   -- output enable
    brdyen  : out std_logic;   -- bus ready
    bexcn   : out std_logic;   -- bus exception (timeout)
    wsen    : out std_logic    -- write strobe enable
  );
end memctrl;
```

The memory map set out in table 4.1 was decided upon for the purposes of this project after taking into account the constraints set out in table 3.1.

Address Range	Size	Mapping	Module
0x00000000 - 0x00003FFF	256 kB	ROM 0	MemCtrl
0x04000000 - 0x04001FFF	128 kB	RAM 0	MemCtrl
0x10000000 - 0x7FFFFFFF	1792 MB	UNUSED	N/A
0x80000000 - 0x8FFFFFFF	256 MB	On-chip Regs	APB Bridge
0x90000000 - 0x9FFFFFFF	256 MB	DSU Regs	DSU

Table 4.1: Device Memory Map

The internal mappings for the on-chip and debug support unit registers had not been altered, so it was not necessary to implement external logic to control their operation. The assignable address space (between 0x00000000 and 0x0FFFFFFF) was initially split into three types of memory access; ROM, RAM and I/O.

```
case addr(26 downto 24) is
    when "000" | "001" => area := rom;    -- 0x00000000 - 0x01FFFFFF
    when "010" | "011" => area := io;     -- 0x02000000 - 0x03FFFFFF
    when others        => area := ram;    -- 0x04000000 - 0x0FFFFFFF
end case;
```

This made it easier to add multiple chip selects for each type of access, even though only one ROM chip and one RAM chip would be used in the design project.

ROM Select

The EPROM that would be used in the project had a capacity of 256kB and this was reflected in the control logic.The device was placed at address 0x00000000 as that was where initial program execution would start after processor reset.

```
    rom0en <= not( (not rin.rom0sn) and (not asn) );    -- active low
```

The above logic ensured that the ROM enable signal is only asserted when there is valid data on the address bus.

RAM Select

The chip select signal for the RAM chip was generated in a similar way to that of the EPROM. It had a capacity of 128kB and was placed at the start of the RAM address space (0x04000000).

```
    ram0en <= not( (not rin.ram0sn) and (not asn) );    -- active low
```

The above logic ensures that the RAM enable signal is only asserted when there is valid data on the address bus.

Output Enable

The output enable signal was generated in such a way as to avoid bus contention between memory mapped devices.

```
    outen <= not( read and (not dsn) );    -- active low
```

This ensured that the device outputs would only be enabled if valid data was present on the data bus and the processor was performing a read cycle.

Write Strobe

The write strobe signal is used to notify peripheral devices if the processor is performing a read or a write cycle. A logic "1" signal means a read cycle and a logic "0" signal means a write cycle.

```
wsen <= not( (not writen) and (not dsn) );    -- active low
```

The above code ensures that a write strobe will only be generated if there is valid data present on the data bus when the processor is performing a write cycle and that all of the bus signals have stabilised.

Bus Ready

The bus ready signal is used to notify the processor that it may end the memory cycle on the next clock cycle. As both the ROM and RAM chips have fast access times, there is no need for a delayed bus ready signal so it is generated as soon as either the ROM or RAM is selected and there is valid data on the address bus.

```
brdyen <= not( ( (not r.rom0sn) or (not r.ram0sn) ) and (not asn) );
```

Bus Error

The bus error signal was tied low for the purposes of testing the processor operation, however a bus timeout signal may easily be generated using an 8-bit counter. It should assert a bus error (timeout) signal after a specified period. This signal may be used to skip or re-initiate the faulting bus transaction.

4.1.4 Simulation

At this point, the control logic had been fully implemented as a VHDL entity, however it's operation had yet to be tested in order to ensure that it would function correctly once synthesised. This was accomplished by creating a new testbench, based on the standard "func8" testbench, that could be used to tie the altered LEON core and control logic module together.

This testbench proved that the control logic operated as expected and that the modified processor code worked and was able to read information successfully from the simulated ROM as well as execute the instructions. The infinite loop test program (detailed in section 5.2) was used at this point in the testbench simulations.

These simulations provided waveforms that could be used to verify the operation of the processor once it was running on-chip, by comparing the expected results against the actual trace results obtained using a logic analyser.

4.2 Hardware Decisions

Several decisions had to be made at this stage, with relation to the hardware that would be used in the project. This included deciding upon which chip package types to use and what kind of signalling standard would be used between the devices. The cost of any required components would have to be evaluated along with their ease-of-use and functionality.

4.2.1 TTL vs LVTTL

The first choice to make was what kind of signalling standard would be used in the new project. The obvious choice was to stay with TTL, as suitable ROM and RAM chips were readily available. This would cut down on the expense of migrating to the new project design due to the fact that it would not be necessary to acquire as many new components. The problem with this approach, however was that the VirtexII FPGA was not capable of fully supporting TTL devices[10].
As a result, the possibility of incorporating logic level convertors into the design of the project was considered. Some devices such as the MAXIM-3000E[11] and the HEF4104B[12] were already available and suitable for use in the project. Although it would be possible to create a system using mixed LVTTL and TTL signalling[13][14], doing so would add an unecessary amount of complexity to the final project design.This would increase the overall time required by the students to build the microprocessor system but would not provide any educational aspect so was discounted.
The VirtexII FPGA was, however, capable of supporting LVTTL[15] signalling. It was decided to implement the entire project hardware using LVTTL components, as it was the most similar standard to TTL available and had been originally created in order to supercede the old TTL technology. This kept in line with the aim of taking advantage of new technological advances during the design of the new microprocessor project.

4.2.2 TSOP vs DIP

In searching for suitable LVTTL components, it became apparant that very few came in DIP packages. This was a major problem, as DIP package component were required if the wire-wrap method was to be used to connect the devices together.
A solution was found when a company was discovered that provided TSOP-to-DIP adapters[16]. These adapters provided a 1-to-1 pin mapping from 32-pin TSOP packages to 32-pin DIP packages. This option was chosen as any ZIF convertor sockets for TSOP-DIP were found to be prohibitively expensive for the purposes of the project.

4.2.3 Interconnects

Components would be connected together using wire-wrapping techniques. This is the same method as is used in the original design project. All of the devices were seated on the breakout area of the VirtexII prototyping board

4.3 ROM

The AT29LV020 is a LVTTL in-system flash Programmable and Erasable Read Only Memory (PEROM)[17]. It was decided to use this device as the ROM due to its availability and the fact that it was already supported by the chip-programming tools used as part of the current design project.
The main features of the device include:

- Single Voltage, Range 3V to 3.6V Supply
- 3V only read and write operation
- Fast read access time (100nS)
- Low power dissipation (15mA active and 40uA Standby)
- Fast programme cycle times
- CMOS and TTL compatible inputs and outputs

The memory capacity of the ROM is 256kB using an 8-bit data bus. This is more than adequate for the purposes of the design project.

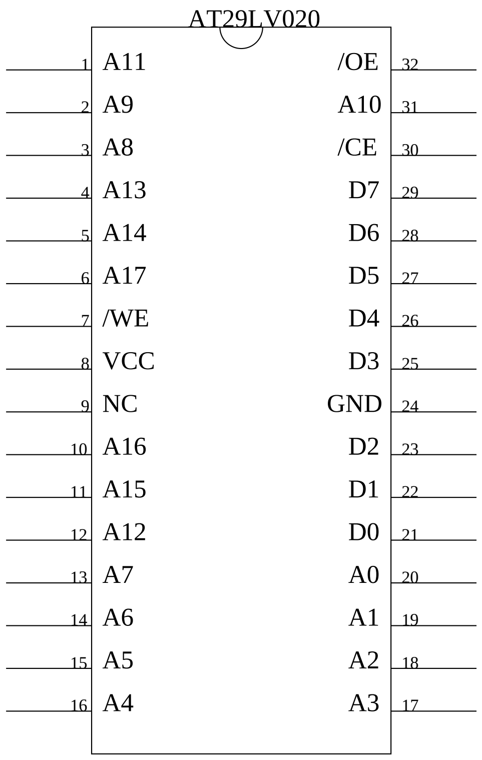

Figure 4.5: Block Diagram of the ROM chip (DIP adapter fitted)

Figure 4.5 shows the pin assignments of the ROM chip when fitted into the TSOP-DIP adapter. The pins are assigned in a 1-to-1 mapping directly from the ROM chip.

4.3.1 Programming the ROM

Even though the ROM chip had been fitted with a DIP adapter, the pin mappings were not correct due to the fact that the adapters provide a 1-to-1 pin mapping between TSOP and DIP format. This is not a problem when designing the project hardware as the pin assignments can be taken into account when connecting the components together. It does, however, become a problem when using the chip-writer to programme the ROM, due to the fact that the pins are mapped in the wrong order for the corresponding DIP version of the chip. This problem was overcome by using the pin mappings in table 4.2 to create a convertor that would alter the pin assignments to make it suitable for use with the chip writing device[1].

TSOP Pin #	DIP Pin #	TSOP Pin #	DIP Pin #
1	25	17	9
2	26	18	10
3	27	19	11
4	28	20	12
5	29	21	13
6	30	22	14
7	31	23	15
8	32	24	16
9	1	25	17
10	2	26	18
11	3	27	19
12	4	28	20
13	5	29	21
14	6	30	22
15	7	31	23
16	8	32	24

Table 4.2: Pin Mappings for Convertor

The pin convertor was made by placing two 32-pin sockets on a small piece of circuit board and then wire-wrapping the connections between the two sockets in the appropriate order according to table 4.2. This pin convertor would be placed between the ROM chip and the chip-writer when it was required to write information to or verify the ROM contents. The ROM chip could be placed directly into the project circuitry without the use of the pin convertor.

With the convertor attached, the ROM chip could be directly programmed using the chip-writer and the "ChipWin" software download programme, in the same way that the EPROMs used in the current design project. The main advantage of the new ROM chip is that it may be electronically erased on the spot before programming it with new information, instead of having to use the UV eraser as with the current EPROM devices.

[1]Obtained from http://www.ebccompany.com/TS32_DRAWINGS.htm

4.4 RAM

The IS63LV1024L is a high speed, low power CMOS static RAM[18]. It was decided to use this device as the RAM due to its availability, 8-bit data bus and LVTTL compatibility.
The main features of the device include:

- High speed access time (12nS)
- High performance, low power CMOS
- /CE power-down function
- Fully static operation
- TTL compatible inputs and outputs
- Single 3.3V power supply

The memory capacity of the RAM is 128 kB using an 8-bit data bus. This is adequate for the purposes of the design project and so it is only necessary to use one RAM chip, cutting down on the amount of wire-wrapping that has to be performed.

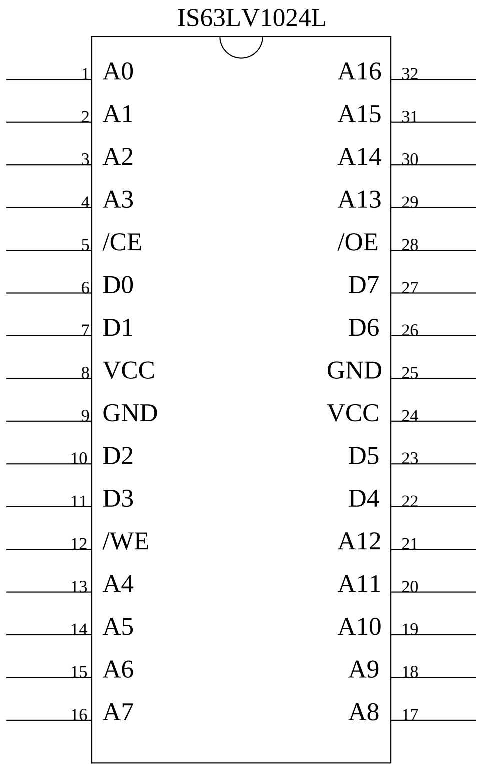

Figure 4.6: Block Diagram of the RAM chip (DIP adapter fitted)

Figure 4.6 shows the pin assignments of the RAM chip when fitted with the TSOP-DIP adapter. The pin assignments are a 1-to-1 mapping directly from the RAM chip. No pin-convertor is required in the use of this device.

4.5 UARTs

For the purposes of the project, the two internal UARTs were left running inside of the LEON core. Both of the UARTs are identical in operation and provide the same functionality as any external UARTs that would be used in the design. The only major difference is that the UART registers are available “on-chip” in the LEON core and are accessed via the internal APB bridge.
The UARTs support data frames in 8-bits, one optional parity bit and one stop bit. The bit-rate can be obtained from either an internal 12-bit clock divider or from an external source, through the parallel interface port. The UARTs support both hardware and software flow control. The UARTs also support loopback mode, whereby the transmitter may be directly connected to the receiver, for the purposes of testing.
The internal UARTs may be individually removed from the LEON core by setting the appropriate options in the graphical configuration utility. If this is the case, external UART devices may easily be incorporated into the hardware system by altering the control logic to add the appropriate chip select signals, mapping the UART registers into an appropriate area of memory, then connecting the chips to the MAX3232 converter in the same way as for the internal UARTs.

4.5.1 LVTTL to RS232

Regardless of whether internal or external UARTs are used in the hardware system, they still required an interface device to translate between LVTTL and RS232 voltage signalling standards. The Maxim MAX3232CPE chip was chosen for this purpose[9] as it is compatible with LVTTL signalling and is readily available in DIP format. The MAX3232 is an LVTTL varient of the MAX232 chip used in the current design project and operates in the same fashion, requiring similar external circuitry for its operation.
Additional circuitry required to work it and is detailed in figure 4.7. This shows the way in which the four external 0.1uF charge-pump capacitors should be connected to the MAX3232.

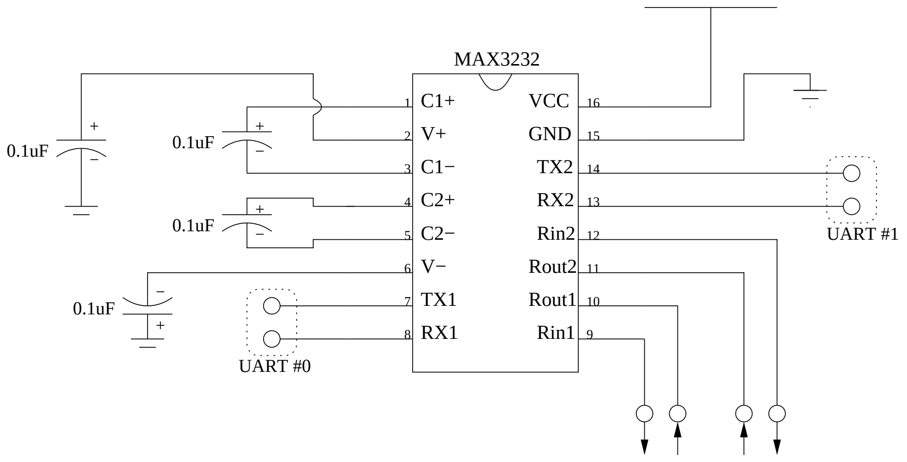

Figure 4.7: Block Diagram of the RS232 Interface Circuitry

Only one MAX3232 is required for the project as it contains two internal receivers and drivers, capable of interfacing the two UARTs to two separate serial ports. An additional MAX3232 would be required, however, if the dedicated DSU serial port is to be used.

4.6 Final Design

The schematic of the final hardware design for the project is given in figure B.1. This details block diagrams of all the different components and the way in which they were connected together to create a fully functional microprocessor system based around the LEON processor.
Figures 4.8 and 4.9 show the final hardware system implemented on the VirtexII prototyping board. The first figure shows the ROM, RAM and MAX3232 on the breakout area, as well as the FGPA and configuration PROM. The second figure shows the wire wrap connections made between the components on the underside of the board.

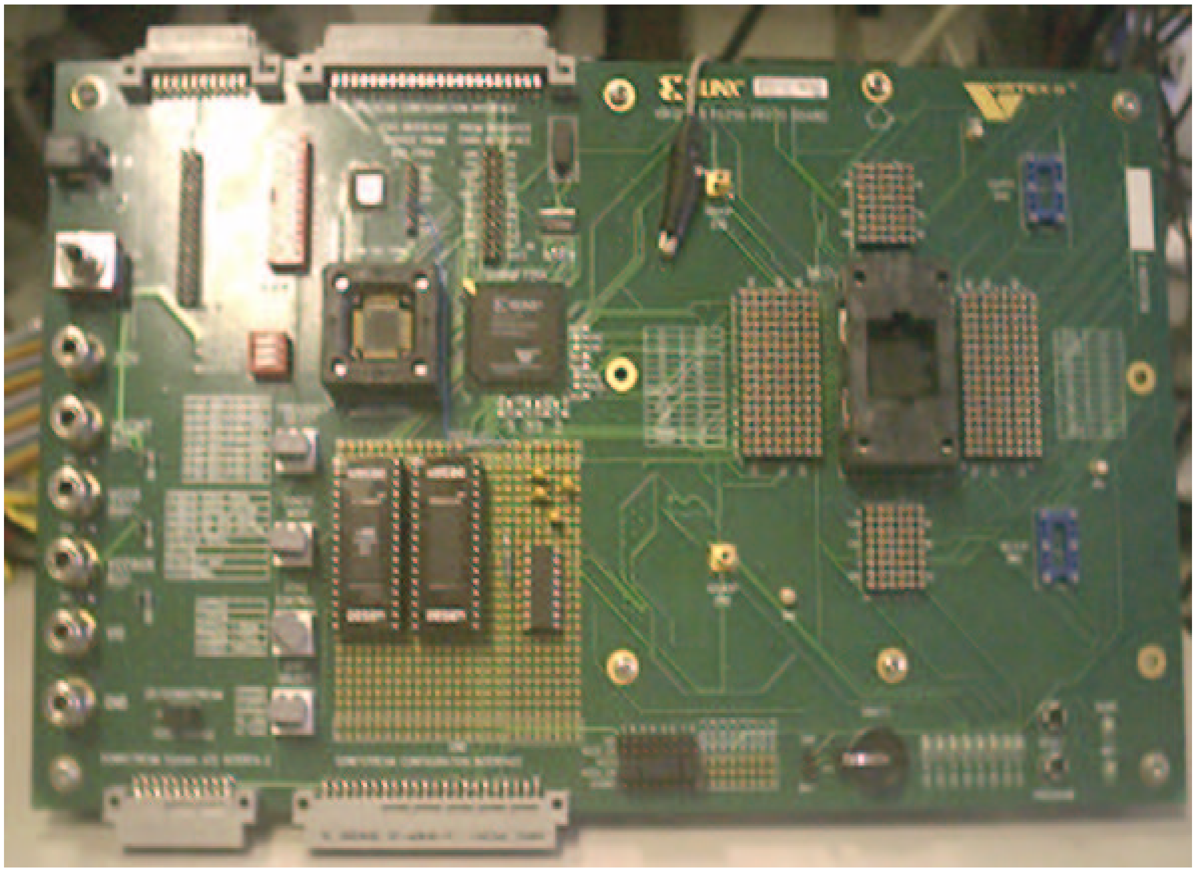

Figure 4.8: Front View of Completed Hardware

In the front view of the prototyping board, the FPGA is the chip situated in the rightmost ZIF socket and the configuration PROM is situated in the ZIF socket directly above the breakout area. The PEROM is the leftmost chip on the breakout area, with the RAM being situated in the center and the MAX3232 and charge-pump capacitors being situated to the right. Both the ROM and RAM chips are shown mounted in their TSOP-to-DIP adapters.

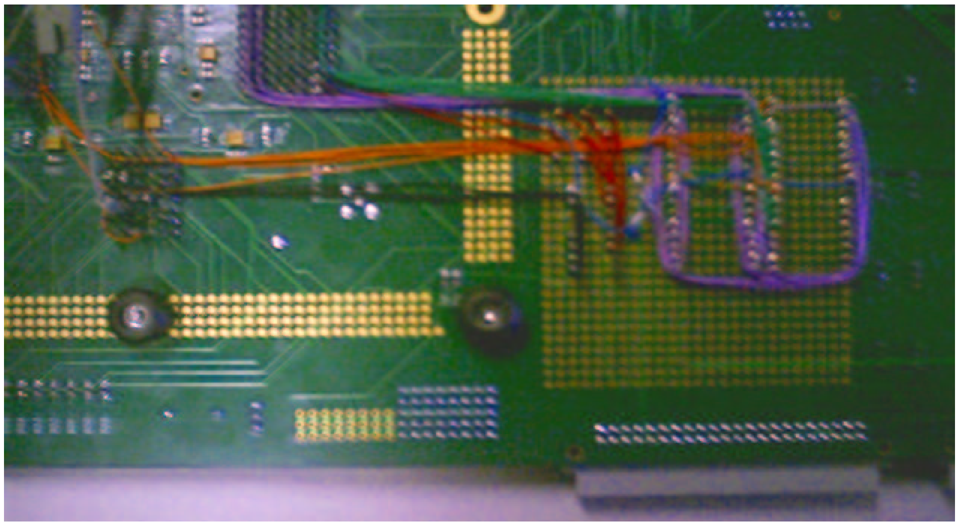

Figure 4.9: Rear View of Completed Hardware

The rear view of the prototyping board details the wire-wrap connections made between the different components. The wires are colour-coded according to their functionality, as given in table 4.3.

Wire Colour	Signal Type
Purple	Address Bus
Green	Data Bus
Orange	Control Signals
Red	Serial Port Wiring
Black	Power and Ground

Table 4.3: Colour coding of wire-wrap connections

Chapter 5

Testing and Conclusions

At this stage, the entire prototype hardware design has been finalized and constructed using the various components and techniques described in chapter 4. The final step in the evaluation of the new microprocessor design project is to test the functionality of the project hardware.
This chapter describes the programs used to test the hardware design and the results obtained, as well as detailing the software tools available for use with the LEON processor and how they may be useful in the project. Finally, the future of this project will be discussed, outlining work that may be continued based on this project.

5.1 Software Tools

Several useful software tools are available free of charge from Gaisler Research and may be obtained from their website. These include a suite of compilation tools for the LEON processor, a simulator suitable for testing new programs without having to put them on-chip and a monitor program, which may be used to interface with the optional internal Debug Support Unit using the dedicated serial port.

5.1.1 The LECCS Compiler

The Leon Erc32 Cross Compiler System (LECCS) allows cross-compilation of assembly code or single and multi-threaded C and C++ applications for the LEON processor. It includes the following components:

- GNU C/C++ Compiler
- Linker, assembler, archiver, etc
- Standalone C-library
- RTEMS real-time kernel
- Boot-prom utility
- Remote debugger monitor for gdb

- GNU debugger with TK frontend
- DDD graphical user interface for gdb

The LECCS compiler was used to compile all of the test programs using the following commands.

```
sparc-rtems-gcc -nostdlib -nostdinc -O2 -Ttext=0 program.S -o program
sparc-rtems-objcopy -O binary program program.bin
```

The `sparc-rtems-gcc` command compiles the assembly, C or C++ program and links it. The `sparc-rtems-objcopy` command then converts the compiled program into binary format. This binary file may then be downloaded directly in binary format onto the ROM, using the chip-writer and download tools.

5.1.2 The TSIM LEON Simulator

TSIM is an instruction level simulator capable of emulating the LEON processor core. The full functionality of the VHDL model is emulated, including caches, on-chip peripherals and memory controllers. The amount of simulated main memory can be configured at run-time.
TSIM can be run in stand-alone mode, or connected to the GNU gdb debugger. In stand-alone mode, a variety of debugging commands are available to allow manipulation of memory contents and registers, break-point insertain and performance measurement. When connected to gdb, TSIM acts as a remote target and supports all gdb debug requests.
The simulator can be obtained for free, under an evaluation licence, as long as it is used for purely academic or educational purposes.

5.1.3 DSUMON

DSUMON is a monitor for the optional LEON processor debug support unit. It supports the following functions:

- Read/Write access to all LEON registers and memory
- Built-in disassembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Remote connection to the GNU debugger
- Auto-probing and initialisation of LEON peripherals and memory settings

It can be run in standalone mode or in conjunction with the GNU gdb. It connects to the LEON debug support unit using the dedicated serial port in the model.

5.2 The Infinite Loop Test Program

The infinite loop program is used to test the basic functionality of the microprocessor. It verifies that the processor is reading information correctly from the ROM and in doing so, also verifys that all of the control logic is operating correctly.

```
.seg "text"
.proc 0
.align 4
.global _start

_start:
    set     0xC0, %g1           82 10 20 C0
    mov     %g1,  %psr          81 88 40 00
    mov     %g0,  %wim          81 90 00 00
    mov     %g0,  %tbr          81 98 00 00
    mov     %g0,  %y            81 80 00 00
    call    main                40 00 00 01
    nop                         01 00 00 00

main:
    jmp     main                81 C0 20 1C
```

The above code is the entire program used to test the operation of the processor. It is written in SPARC assembly language and compiled using the LECCS cross compiler for LEON. It was downloaded into the ROM chip using the WinChip download program and chip-writer as described in chapter 4.
The infinite loop program has two parts. The `_start` section initialises the processor status register, the trap base register, the "y" register and the window invalid mask register and then calls the main part of the program. The `main` section enters the processor into an infinite loop, continually calling itself.
The numbers to the right of the code show the hexidecimal values of the binary bit-stream of code that was downloaded onto the ROM chip. This may be used to verify the functionality of the processor as it should be possible to view "81 C0 20 1C" being repeatidly fetched by the processor from the memory.
The infinite-loop program can also be used to verify that the instruction and data caches have been removed from the processor correctly. If they were functioning, the processor would not need to keep fetching the `jmp main` line from the memory, but instead would retrieve it from the internal cache. If this were the case, no activity would be present on the address and data buses. If however, the caches were removed correctly, the address and data bus would continuously show the processor retrieving the appropriate information from the memory. This can easily be verified to be the case using a logic analyser connected to the address and data buses.

5.2.1 Test Results

The results of running the infinite loop program on the physical processor showed that everything worked perfectly first time...

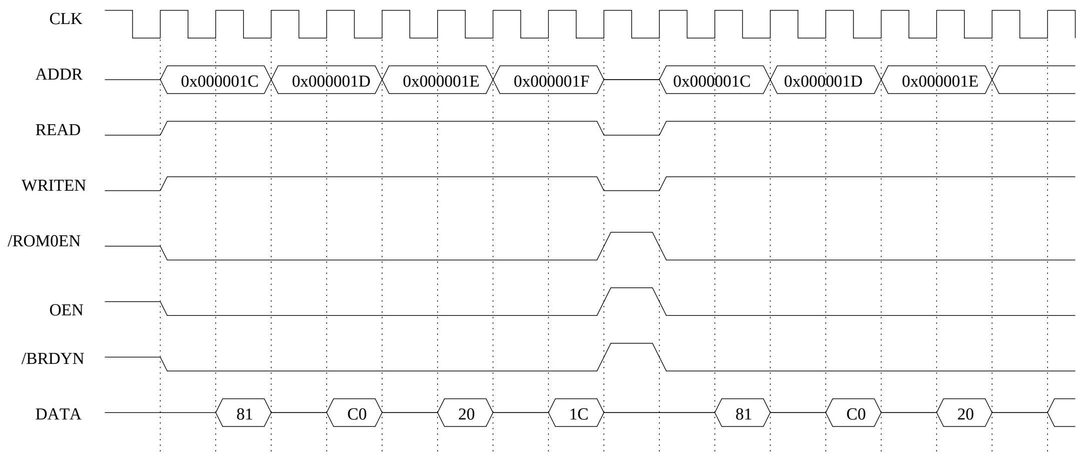

Figure 5.1: Expected Waveform Results of Infinite Loop Program

Figure 5.1 shows the waveform obtained from simulations running the infinite loop program. The operation of the processor was verified by the trace results obtained from the logic analyser, shown in figure 5.2.

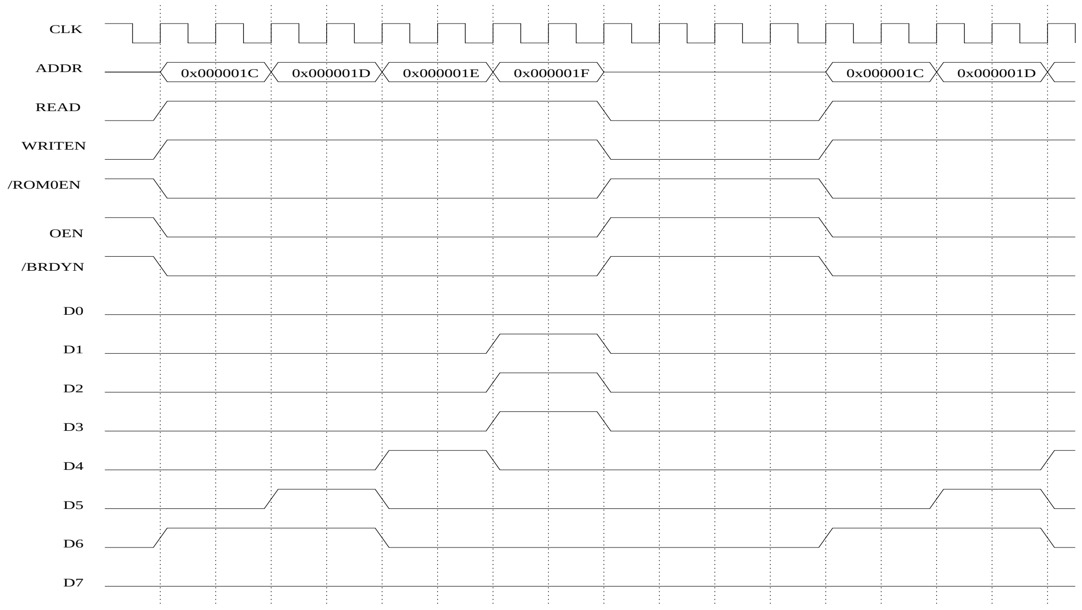

Figure 5.2: Actual Waveform Trace from Logic Analyser

This test proved the operation of the processor. It was continuously reading the same data from the memory and proved that the caches had been disabled correctly.

5.3 Conclusions

During the course of this project, it was determined that the LEON processor is indeed suitable for use as a teaching-aid and would be able to replace all of the functionality of the Motorola MC68008 processor. The LEON processor was modified in order to make it suitable for use in place of the Motorola processor and a new design project was created and implemented to replace the current one.
This new design project was modelled closely on the format of the original project and achieved the aim of creating an updated project, designed around an FPGA based microprocessor, while still keeping to a format that allows students to learn about computer architecture.
This project succeeded in it's aims, resulting in a fully configurable implementation of the LEON core and a new microprocessor design project layout, which is similar in format and implementation to that of the current design project yet flexible in nature due to the underlying hardware.

5.4 Future Work

The work carried out during the course of this project is the first step towards implementation of a new microprocessor design project. The basic hardware design requirements and components have been set out and its operability proved, however further projects arise from this work. These include:

- Designing a dedicated project board
- Writing the transparent link and a monitor program in SPARC assembler code
- Implementing several different LEON configuration options on the one board
- Evaluating different processor architectures, which may be implemented using FPGAs
- Implementing a synthesisable model of an MC68008 in VHDL

These projects would help to build on the work accomplished in this project and realize the full potential of a microprocessor design project which is based on a reconfigurable hardware platform.

Appendix A

Demo Program

```
module demo (clk, led, USER_RESET);

output [7:0] led;
input        clk;
input        USER_RESET;

reg [7:0]  led;
reg [20:0] count;
reg [20:0] count2;
reg [8:0]  shift;
wire        USER_RESET;
reg        nxt_count;
wire clk;

always @(posedge clk)  begin
    if (USER_RESET) begin
        count  <= 0;
        nxt_count  <= 0;
        shift  <= 1;
        count2 <= 0;
    end
    else begin
        if (nxt_count == 0) begin
            count <= count + 1;
            if (count == 100000) begin
                shift  <= shift << 1;
                led[0] <= shift[0];
                led[1] <= shift[1];
                led[2] <= shift[2];
                led[3] <= shift[3];
                led[4] <= shift[4];
                led[5] <= shift[5];
                led[6] <= shift[6];
                led[7] <= shift[7];
                if (shift[8] == 1) begin
                    count     <= 0;
                    count2    <= 0;
                    nxt_count <= 1;
                    shift     <= 1;
                end
            end
        end
        else if(nxt_count == 1) begin
            count2 <= count2 +1;
            if (count2 == 100000) begin
                shift  <= shift << 1;
                led[7] <= shift[0];
                led[6] <= shift[1];
                led[5] <= shift[2];
                led[4] <= shift[3];
                led[3] <= shift[4];
                led[2] <= shift[5];
                led[1] <= shift[6];
                led[0] <= shift[7];
                if (shift[8] == 1) begin
                    nxt_count <= 0;
                    count     <= 0;
                    count2    <= 0;
                    shift     <= 1;
                end
            end
        end
    end
end
endmodule
```

The above code can be used to test the functionality of the VirtexII FPGA and prototyping board. It is written in Verilog and its function is to alternately flash the on-board LEDs continuously. Table A.1 details the pin assignments in the user constraints file used.

Signal	LOC #	Signal	LOC #
clk	R8	USER_RESET	T13
led_0	P13	led_1	R13
led_2	N12	led_3	P12
led_4	P5	led_5	N5
led_6	R4	led_7	P4

Table A.1: Pin Constraints used for the Demo Program

Appendix B

LEON

B.1 LEON Configuration File

The following code is taken from the *device.vhd* file and is used to set the configuration options for all of the modules within the LEON core. It details the set-up used for the purposes of synthesising LEON during the project. The target technology is set for the VirtexII chip, the caches have been disabled, the memory control signalling is set to emulate the MC68008 and the reset generator is disabled. The two internal UARTs are left enabled and the expected system clock signal is set to 6.25 MHz.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.target.all;

package device is

  constant syn_config : syn_config_type := (
    targettech => virtex2, infer_pads => false, infer_ram => false,
    infer_regf => false, infer_rom => false, infer_mult => false, rftype => 1);

  constant iu_config : iu_config_type := (
    nwindows => 8, multiplier => none, mulpipe => false, divider => none,
    mac => false, fpuen => 0, cpen => false, fastjump => false,
    icchold => false, lddelay => 1, fastdecode => false, watchpoints => 0,
    impl => 0, version => 0, rflowpow => false);

  constant fpu_config : fpu_config_type :=
    (core => meiko, interface => none, fregs => 0, version => 0);

  constant cache_config : cache_config_type := (
    enable => false,
    isets => 1, isetsize => 2, ilinesize => 4, ireplace => rnd, ilock => 0,
    dsets => 1, dsetsize => 1, dlinesize => 4, dreplace => rnd, dlock => 0,
    dsnoop => none, drfast => false, dwfast => false);
  constant ahbrange_config  : ahbslv_addr_type :=
        (0,0,0,0,0,0,0,0,1,7,7,7,7,7,7,7);

  constant ahb_config : ahb_config_type := ( masters => 1, defmst => 0,
    split => false, testmod => false);

  constant mctrl_config : mctrl_config_type := (
    memsel => false, bus8en => true, bus16en => false, wendfb => false,
    ramsel5 => false, sdramen => false, sdinvclk => false);

  constant peri_config : peri_config_type := (
    cfgreg => true, ahbstat => false, wprot => false, wdog => false,
    irq2en => false, ahbram => false, ahbrambits => 11,
    reseten => false, uart1_en => true, uart2_en => true);

  constant debug_config : debug_config_type := ( enable => true, uart => false,
    iureg => false, fpureg => false, nohalt => false, pclow => 2,
    dsuenable => false, dsutrace => false, dsumixed => false,
    dsudpram => false, tracelines => 64);

  constant boot_config : boot_config_type := (boot => memory, ramrws => 0,
    ramwws => 0, sysclk => 6250000, baud => 19200, extbaud => false,
    pabits => 11);

  constant pci_config : pci_config_type := (
    pcicore => none, ahbmasters => 0, ahbslaves => 0,
    arbiter => false, fixpri => false, prilevels => 4, pcimasters => 4,
    vendorid => 16#0000#, deviceid => 16#0000#, subsysid => 16#0000#,
    revisionid => 16#00#, classcode =>16#000000#, pmepads => false,
    p66pad => false, pcirstall => false);

  constant irq2cfg : irq2type := irq2none;

end;
```

B.2 Pin Constraints

Table B.1 details the signal-to-pin mappings used in the project. Pins pio_0, pio_1 and pio_2 should have a "PULLDOWN" resistor attached and pins pio_10 and pio_11 should have a "PULLUP" resistor attached. This sets the processor up for 8-bit external PROM mode, accepting an external source on pio_3 as the UART baud reference signal (according to table 2.3).

Signal	LOC #	Signal	LOC #	Signal	LOC #
clk	R8	clkout	R10	clkuart	R11
resetn	T13	errorn	P13	wdogn	R13
read	A12	writen	E15	asn	B12
dsn	B13	iosn	C16	oen	C12
romen	C13	ramen	E14		
brdyn	D12	bexcn	D13		
address_0	A6	address_1	A7	address_2	A8
address_3	A9	address_4	A10	address_5	A11
address_6	B6	address_7	B7	address_8	B8
address_9	B9	address_10	B10	address_11	B11
address_12	C6	address_13	C7	address_14	C8
address_15	C9	address_16	C10	address_17	C11
address_18	D6	address_19	D7	address_20	D8
address_21	D9	address_22	D10	address_23	D11
address_24	E6	address_25	E7	address_26	E10
address_27	E11				
data_24	E4	data_25	E3	data_26	E2
data_27	E1	data_28	D5	data_29	D3
data_30	D2	data_31	D1		
pio_0	M6	pio_1	M7	pio_2	T8
pio_3	T9	pio_4	M10	pio_5	M11
pio_6	N6	pio_7	N7	pio_8	N8
pio_9	N9	pio_10	N10	pio_11	P11
pio_12	P6	pio_13	P7	pio_14	P8
pio_15	P9				

Table B.1: User Constraints Pin Allocations

B.3 Project Board Circuit Diagram

Figure B.1 is a schematic of the final design for the hardware that was used in the project. It details the control logic block as an external entity and shows all of the interconnects required to make the LEON processor operate correctly. The ROM and RAM chips are shown in DIP packaging, as appear in the actual hardware system.

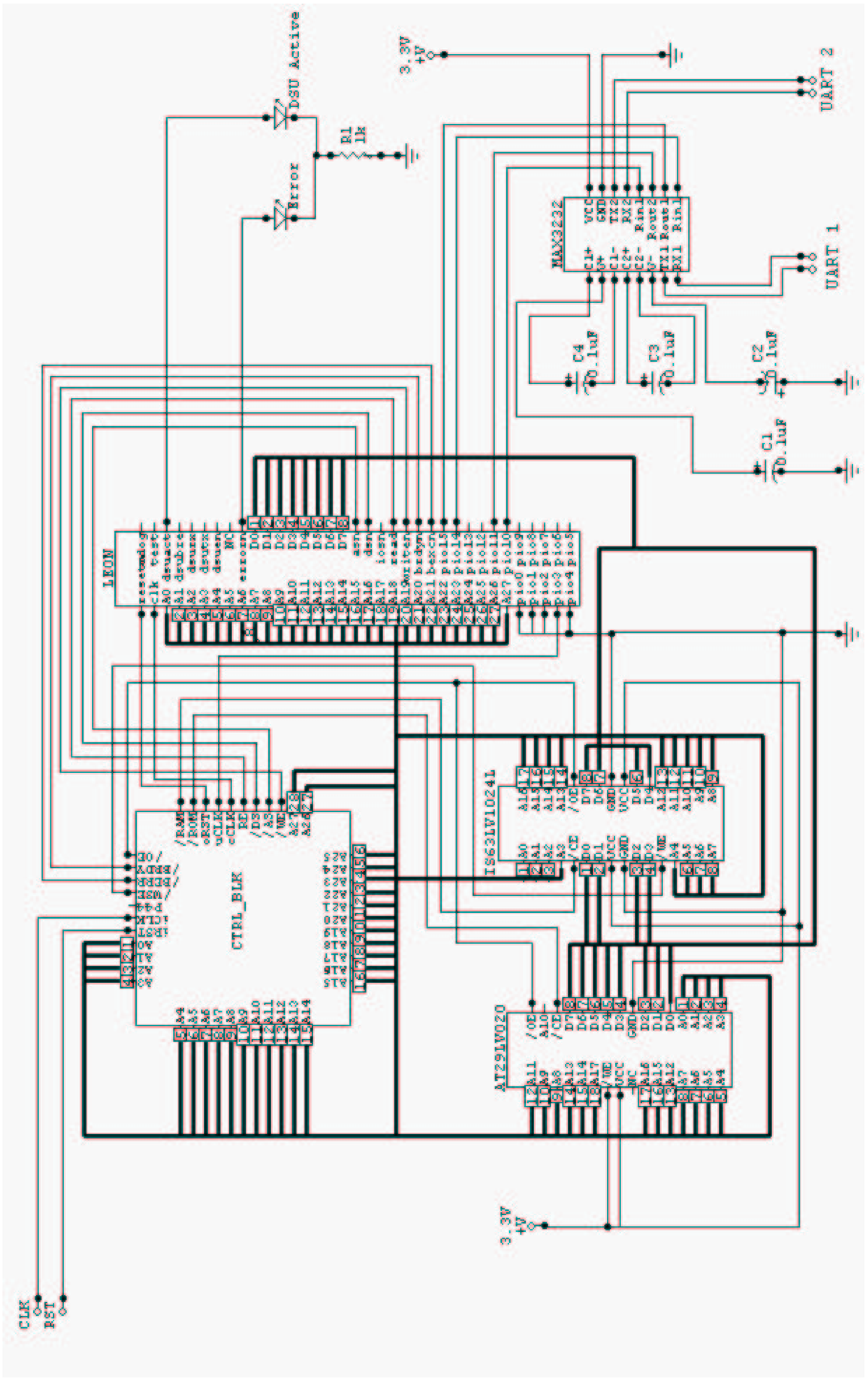

Figure B.1: Circuit Diagram of the Project Hardware System

Bibliography

[1] Murray Pearson, Dean Armstrong, Tony McGregor, "Using Custom Hardware and Simulation to Support Computer Systems Teaching," *Workshop on Computer Architecture Education*, 2002.

[2] Daniel Ellard, Daivd Holland, Nicholas Murphy, Margo Seltzer, "On the Design of a New CPU Architecture for Pedagogical Purposes," *Workshop on Computer Architecture Education*, 2002.

[3] Xilinx, *Introduction to the VirtexII Product Family.* Xilinx Inc, December 2001. http://www.xilinx.com.

[4] Xilinx, *VirtexII Prototype Platform Users Guide.* Xilinx Inc, June 2001. http://www.xilinx.com.

[5] Xilinx, *XC18V00 Series Programmable Configuration PROMs.* Xilinx Inc, November 2002. http://www.xilinx.com.

[6] SPARC, *SPARC V8 Manual.* SPARC International Inc, January 1992. http://www.sparc.org.

[7] J. Gaisler, *Leon2-1.0.10 Users Guide.* Gaisler Research, December 2002. http://www.gaisler.com.

[8] ARM, *AMBA Specification V2.0.* ARM Limited, May 1999. http://www.arm.com.

[9] Maxim, "Maxim MAX3232 Transceiver," tech. rep., Maxim Integrated Products, June 1996. http://www.maxim-ic.com.

[10] Xilinx, "5V Tolerent I/Os," tech. rep., Xilinx Inc, May 2002. http://www.xilinx.com.

[11] Maxim, "Maxim 8-Channel Level Translators," tech. rep., Maxim Integrated Products, October 2002. http://www.maxim-ic.com.

[12] "Quadruple Low to High Voltage Translator with 3-state Outputs," tech. rep., January 1995.

[13] K. Ristow and S. Perna, "Mixed 3.3V and 5V Systems with LVT Logic," tech. rep., Texas Instruments Incorporated, July 1994. http://www.kmeif.pwr.wroc.pl.

[14] J. F. Wakerly, “Low-Voltage CMOS Logic and Interfacing,” 1999.

[15] Xilinx, *VirtexII Advance Product Specification.* Xilinx Inc, September 2002. http://www.xilinx.com.

[16] W. Adaptics, “SOP to DIP Generic Adapters,” tech. rep., Winslow Adaptics. http://www.winslowadaptics.com.

[17] Atmel, “Atmel AT29LV020 Flash Memory,” tech. rep., Atmel Corporation, May 2002. http://www.atmel.com.

[18] ISSI, “ISSI IS63LV1024L Static RAM,” tech. rep., Integrated Silicon Solutions Inc, July 2002. http://www.issi.com.