Synthesisable VHDL model of a Java Virtual Machine for HAL

Louise Annamarie Reilly
Supervised By: Michael Manzke
B.A. (Mod.) Computer Science
Final Year Project

May 2005

Abstract

The aim of this final year project was to implement a Java Virtual Machine on an FPGA. The FPGA is located on a hardware learning board used by second year Computer Science and third year Engineering students as part of their coursework. The introduction of a JVM to this hardware will enable students to program the FPGA with Java in place of lower level languages such as Assembly or VHDL, VHDL being the language currently in use.

This project is part of a continuing effort to upgrade the learning boards from the Motorola MC68008 microprocessor technology which has been part of the course for many years. It built upon the final year project of Ross Brennan. His project involved implementing a RISC microprocessor in place of the CISC architecture which was there and resulted in a prototype project board being designed and assembled. It was one of these boards which was used here.

The JVM implemented on the upgraded board originated as part of a PhD thesis at the Technical University of Vienna, Austria. It is an optimized JVM designed specifically for Altera software and the Cyclone board. As part of the project the original VHDL code was altered to run on Xilinx software and the Virtex-II board which the students will be working with[*][†].

[*]All typesetting for this report was done with LaTeX tools

[†]All diagrams, except *Figure 4.1*, in this project were produced using Photoshop 7.0

Contents

1 Introduction **5**

2 Background & Research **8**
- 2.1 Background of the Hardware 8
- 2.2 Background of the Software 9
  - 2.2.1 JVM Basics 9
  - 2.2.2 JOP Basics 13

3 JOP Design **15**
- 3.1 Breaking Down The Code 15
- 3.2 The Core 15
  - 3.2.1 Fetch component 16
  - 3.2.2 Bytecode Fetch component 17
  - 3.2.3 Decoder 17
  - 3.2.4 Stack 17
- 3.3 Extension 18
  - 3.3.1 Multiplier 19
- 3.4 IO components 20
  - 3.4.1 Universal Asynchronous Receiver-Transmitter 21
  - 3.4.2 Clock Counter 22
- 3.5 External Memory 22

4 JOP Implementation **23**
- 4.1 Generic Parameters 23
- 4.2 Altera Library Files 24
- 4.3 Other Methods 24
- 4.4 Core 24
  - 4.4.1 BCFetch 25
  - 4.4.2 Stack 25
- 4.5 External Memory 25
- 4.6 Short RAM 26

4.7 Clock Divider 27
4.8 Top Level of Design 28
4.9 Pipeline Architecture 28

5 Hardware Analysis **31**
5.1 Hardware Description 31
5.1.1 Configuration Modes 32
5.1.2 JTAG Mode 32
5.1.3 Slave Serial Mode 34
5.2 FPGAs in Detail 34
5.3 Putting the JVM onto the FPGA 35
5.4 Additional Problems 36

6 Evaluation & Results **38**
6.1 Stack Testing 38
6.2 Fetch Testing 40
6.3 BCFetch & Decode Testing 40
6.4 Core Testing 41
6.5 Extension Testing 43
6.6 External Memory Testing 44
6.7 IO Testing 45
6.8 Set RAM Testing 45
6.9 8bit RAM Testing 46
6.10 JOP Testing 48

7 Future Work & Conclusions **49**
7.1 Future Work 49
7.1.1 Class Loader 49
7.1.2 Compiler 49
7.1.3 Editor 50
7.1.4 GUI 50
7.1.5 External Memory 50
7.1.6 Threads, Interrupts and Scheduling 51
7.2 Conclusions 51

A ROM **55**

B RAM **57**

List of Tables

3.1 Mapping of bytecode to JVM address 18
3.2 JOP hardware registers and memory areas 19
3.3 Address mapping for Extension 20
3.4 Operations of a Booth Multiplier 20
3.5 Address mapping for IO 21
3.6 Memory mapping for External Memory 22

4.1 Binary bits of count zero to seven 27
4.2 5-stage pipeline 28

5.1 Configuration Mode 33
5.2 JTAG Mode 34
5.3 Parallel Cable connections 35

List of Figures

2.1 JVM Platform 10
2.2 JVM Components 11

3.1 Components of the JVM 16
3.2 Receive and Transmit Buffers 21

4.1 JOP Datapath 30

5.1 Virtex II Prototype and Demonstration Boards 33

6.1 Stack Testbench Waveform 39
6.2 Fetch Testbench Waveform 40
6.3 BCFetch Testbench Waveform 41
6.4 Decoder Testbench Waveform 42
6.5 Core Testbench Waveform 43
6.6 Extension Testbench Waveform 44
6.7 External Memory Testbench Waveform 45
6.8 IO Testbench Waveform 46
6.9 RAM Setter Testbench Waveform 46
6.10 New RAM Testbench Waveform 47

Chapter 1

Introduction

Throughout the course of this report, it is hoped that the reader will gain a firm understanding of the methods required to:

1. Create a Java Virtual Machine.
2. Implement a Java Virtual Machine on a microprocessor.
3. Run programs on an FPGA.
4. Port VHDL code from Altera to Xilinx.
5. Test VHDL code.

These five processes will be covered in detail in the coming chapters.

This project set out originally to implement VHDL code for a JVM[*] on a piece of available hardware referred to as the HAL[†]. It builds upon the work of Ross Brennan two years ago, which began the HAL project.

There is always a need to expand knowledge and to ascertain what is possible. This project is a good representation of this, being a research one. It tests the boundaries of an FPGA[‡] on board the HAL. At the beginning of it, no-one was even certain if placing a JVM on the hardware board was possible. Much time was expended researching similar projects as well as working on the JVM at hand.

The HAL project is currently trying to have several CPUs on the one board. The FPGA is easily configurable and can be reprogrammed in very little time. Therefore it has the potential to run several different functions

[*] Java Virtual Machine
[†] Hardware and Architecture Lab-board
[‡] Field Programmable Gate Array

and have many different CPUs all providing different operations and modes of use. There are several projects which could span from Ross', it has opened the door to many new possibilities. Putting a Java platform on the board is just one of these possibilities and has been attempted in this project.

The HAL (Chapter 2) is a learning board for Engineering and Computer Science students of Trinity College Dublin. The boards which currently make up part of the Trinity courses are different to the one used in this project. The older boards draw upon technology from the early 80's and are now almost totally obsolete. It is hoped that they will be changed over to the new HAL boards. The HAL board is a Virtex II Prototype Platform board which was altered by Ross Brennan in his Final Year Project[3] to replace the older boards.

The major problem faced when undergoing such a project as this is lack of knowledge. Chapter 2 not only details the hardware board utilised in this project, it also covers a lot of the research done to try to gain a firm grasp of what is involved in implementing a JVM written for a different hardware board on the available one. A synopsis of the research involved in this project is included in this chapter. It will go through the basics of JVMs in general and the JOP[§], Martin Schoeberl's version of the JVM[14], in particular. The background of the VHDL code which produces the JVM is gone through here. This code was downloaded from Schoeberl's website. Much time was spent reading through documentation on this website as well as that of other sources.

Chapter 3 covers the original VHDL code downloaded from Schoeberl's site[14]. The code has been split modularly to better explain what each section does. Having analysed each integral code segment their interactions with each other are examined. This will provide the reader with a clear picture in their mind of how the JOP was originally written and how JVMs generally interact with each other. In doing this the reader is provided with the necessary prerequisite knowledge of the JOP and an understanding of the foundation on which the project has been based.

The alteration of the design and subsequently coded implementation is detailed in Chapter 4. The code had to be ported from Altera on Cyclone boards to Xilinx running on Virtex II boards as the original software and hardware platforms were unavailable for this project. In addition, several sections of new code had to be introduced which necessitated fundamental changes to the circuit design and subsequently its component interactions. This chapter will go over such changes and shows much of the work undertaken.

[§]Java Optimized Processor

Chapter 5 gives physical details of the HAL. Included in this chapter is a full description of the layout of the board and its different modes of operation. This chapter will go into detail on how to use the board and there is a summary of everything needed to program the HAL with VHDL code. This was a significant component in the final stage of this project and would ultimately provide proof of concept. As such great care was taken in the correct operation of the software and hardware involved in coding and testing. This is where the completed JVM would run. The following two chapters will cover all of the elements of this stage in the project with the latter describing the testing which was required on the board.

Chapter 6 describes what was involved in the overall testing and evaluation. The code was tested extensively with ModelSim simulation tools before finally being loaded onto the board and tested there. Each component of the JOP had to be tested. The evaluation was an ongoing process and as such was dispersed throughout the duration of this project. At every point, from first downloading the code, to getting it onto the board, there were tests carried out to ensure that everything was operating as it should be. The testing finally ended when the JOP was running correctly on the board and, as such, this chapter completes the implementation process of the JOP. After reading these chapters the reader should have a firm grasp of the different processes undertaken to complete this project.

Chapter 7 suggests several avenues which may be pursued as a continuation of this project. In this chapter the reader will find several areas where the study of JOP in this environment can be built on. There are feature recommendations as well as full projects which could spawn from the work to date and the chapter covers a few of the possibilities.

Chapter 2

Background & Research

2.1 Background of the Hardware

As has already been mentioned, this project builds upon work done by Ross Brennan[3] two years previously for his final year project. His project involved replacing the MC68008 microprocessor with a LEON2-1.0.10-xst* SPARC-V8 processor implemented on an FPGA. The LEON processor was freely released under the GNU-LGPL license and is available for download from the Internet.

In the process of testing the new processor, Ross designed two boards capable of running and using it. He used the Virtex II prototyping and design boards available from Xilinx. On these he connected various chips, including an Atmel AT29LV020 ROM, an ISSI IS63LV1024L RAM and a Maxim-IC MAX3232CPE IO chip. Currently, it is hoped that these boards may eventually be capable of running several different CPUs. This project, Ross', and several others are being attempted towards this end.

The original MC68008 boards were used as teaching aids for second year Computer Science and third year Engineering students in Trinity College Dublin. The students are able to get a hands-on feel for microprocessor design by wiring up the MC68008 boards, connecting the microprocessor to external chips, and programming them. The microprocessors on these boards are programmable using Assembly language and are implemented using CISC† architecture. Though still a worthwhile exercise, these boards are by now severely outdated and in need of upgrading. Hence the use of final year projects, such as Ross' and mine, to find a newer solution than the MC68008.

*available from http://www.gaisler.com

†Complex Instruction Set Computer

The LEON core is an example of a RISC[‡] architecture. Chips designed using RISC architecture are cheaper to design and run quicker than those implemented in CISC. CISC use more complex instructions, these can be slow and time consuming on a processor. On the other hand, RISC have relatively few instructions, and those it does have are kept simple and brief, making them quicker to run. This makes RISC, in places, more advantageous than CISC.

One of these places is on the learning boards. The students use these boards to get an insight into architecture design. They concentrate more on the hardware than the software implemented on the boards and don't need as many instructions.

2.2 Background of the Software

The original code for the JVM was downloaded as open source code from http://www.jopdesign.com[14]. This site holds work and documentation completed by Martin Schoeberl as part of his PhD. thesis in the Technical University of Vienna, Austria. He set out to develop a simple Java processor which was optimized down so that it only executes Java bytecode[14]. He called his hardware implementation of this processor JOP.

2.2.1 JVM Basics

The Java Virtual Machine allows a Java program to be run on any hardware. Many programming languages are written to be run on a particular Operating System. Java however, was released as an integrated language for the Netscape Internet browser[14]. This meant that at any given time it could be run on any Operating System. Java needed to be portable.

The JVM provides this portability. A JVM resides on the hardware, it interacts with the OS in native code belonging to that system. Any Java programs trying to run on the OS must work through the JVM. It is the Java translator.

The JVM can be split up into several different layers, as shown in *Figure 2.1*[13]. The uppermost layer is the part which the programmer or user interacts with. It holds the actual Java Application and generally comes in the form of an editor or compiler, or both, which the JVM provides. The programmer puts his/her code into this upper layer and compiles it.

When the program is compiled, a .class file is created. These .class files contain:

‡Reduced Instruction Set Computer

1. Virtual machine code for methods of the Java class.
2. Symbolic reference to the upper levels of the Java class.
3. List of fields defined by the Java class.
4. Literals and symbols used by the Java class.
5. Other data required by the Java runtime system.

What this boils down to is that the .class file contains bytecode[6]. The original code is broken down and translated to this. Bytecode is an intermediate code which exists between the source and machine code[9]. The bytecode encodes the above list, the JVM then translates the .class file to a .exe[§] file[6], and runs it on the lower down Operating System, thus providing Java with much of its portability and basically acting as a translator or a go-between.

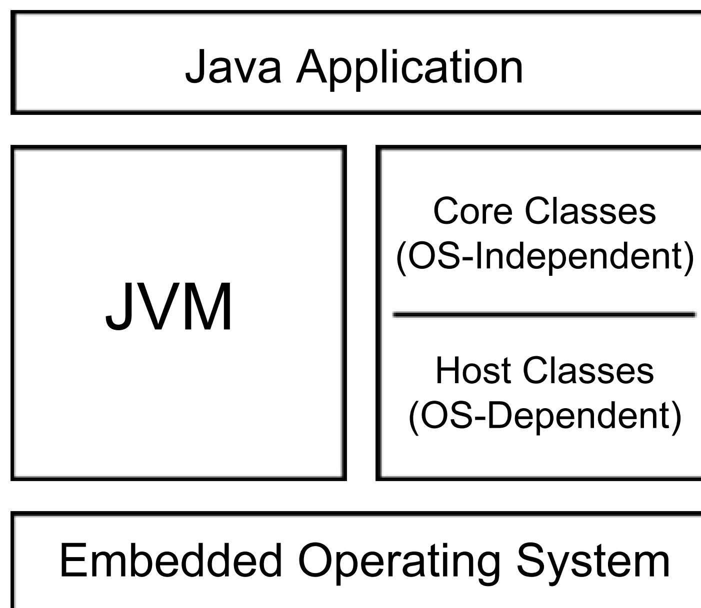

Figure 2.1: JVM Platform

The JVM translates the bytecode to machine code, specific to whichever machine the JVM currently rests on, or rather, the JVM interacts between classes from the Java Application and those residing in the Operating System. It provides bytecode which is an intermediate language between them and

[§]Executable

acts as a medium, thereby running the Java classes on whatever hardware is available.

Figure 2.2 shows a more in-depth view of the components of a basic JVM[9]. This figure can be related to the previous one, but provides greater detail of the components and how the JVM works.

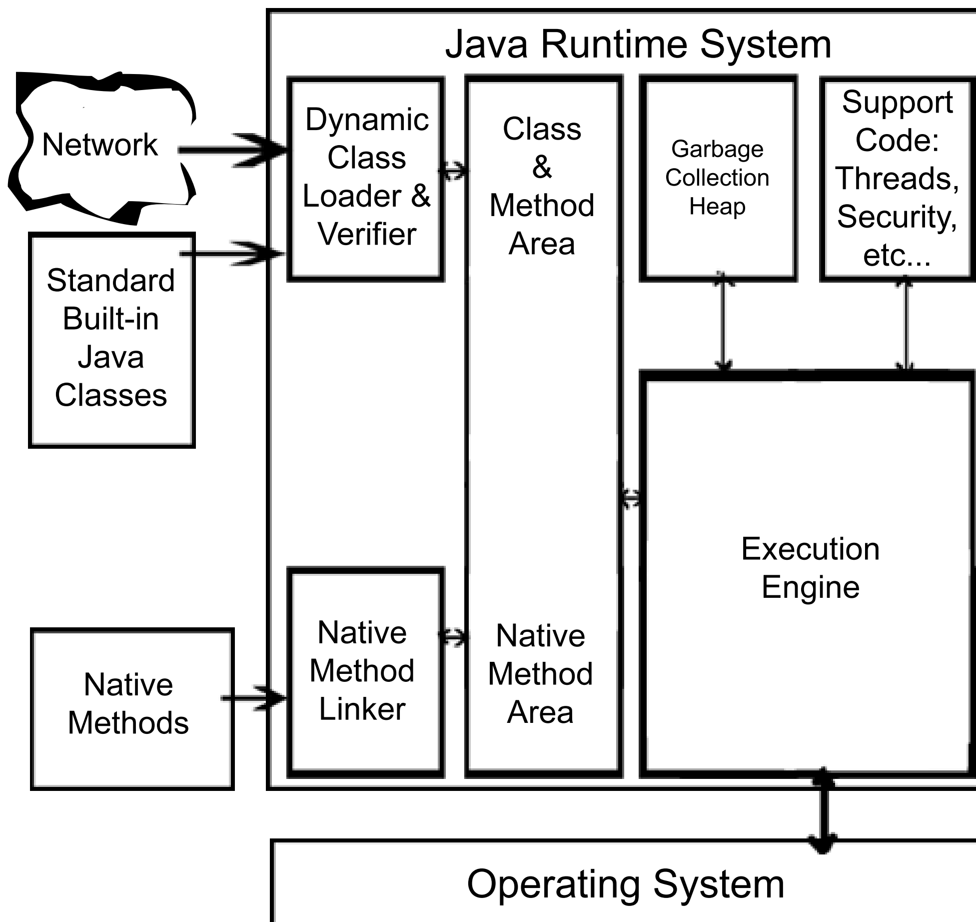

Figure 2.2: JVM Components

The Java Runtime System is the JVM.

The Execution Engine is the equivalent of a virtual processor[9]. It receives a stream of bytecode and executes instructions contained therein. It interacts directly with most of the JVM as well as with the Operating System.

Native Method Support enables calling of methods from the underlying code in different languages. It is the translator from bytecode to machine code and is hardware specific. Referring to *Figure 2.1*, it is the link between the core and the host classes. The native methods are those belonging to the

OS, generally C/C++. The JVM associates these with the bytecode instructions contained in the Execution Engine and thus translates the bytecode to its corresponding native method, ie: creating a .exe file, and sends them to the Operating System.

The Class Loader dynamically loads Java classes from the Java class files[9]. These files can come from Java Library files or from the programmer. The Class Loader quickly runs through and verifies the code, preparing the files for execution.

The Class and Method Area holds necessary bytecode for programs. It has areas for both native methods and Java classes, feeding both through to the Execution Engine as they are needed.

The Garbage Collection Heap reclaims system resources when the code has finished using them[13]. It simplifies programming, helping to avoid such classic errors as memory leaks.

Support Code controls threads, interrupts, exceptions, etc. Threads and scheduling tools in Java have very loosely defined rules. For example, a lower priority thread can preempt a higher priority one. This does help prevent starvation but unfortunately is unacceptable in a JVM due to the importance of interrupts to a real-time system. The support code helps alter this behaviour[14].

The JVM simply acts as a platform between the Java program and the Native Methods of the Operating System. JVMs are OS specific, their lower layers interacting directly with the OS. Their upper levels however, can run any Java software. Therefore, a program written on a Mac, could be transported to a machine running Linux, and from there to one running Windows with no problems. This is because the Java program would not have to change at all to compile on the JVM on the system. The system JVM would act as a cushion between the program and the OS, translating easily from source to machine code for any of the machines and any Operating System.

Other JVMs

There are several other JVMs available. During the research section of this project, many were looked at. They provided a wider perspective of the project and how it can be implemented, as well as an insight into the amount of work required to get even the basic features working.

One such was *kissme*. *Kissme* is freely available under the GNU[¶] GPL[‖]. It was an Honours project by Stephen Tjasink in the University of Cape Town, South Africa. It was developed for a set-top box that came with a TV

[¶] GNU's Not UNIX

[‖] General Public License

Satellite Decoder. The JVM was run on a slow processor and class files had to be stored in the ROM, similar to the JOP. This project was continued as part of Stephen Tjasink's Masters degree to support more advanced features such as persistent programming[7].

Kissme works on UNIX systems and can run console Java applications. It is written in C, Assembly and Java. It currently has native threads, garbage collection and uses GNU classpath libraries. It supports an extension for orthogonal persistence which will transport Java data structures between memory and a persistent store, a feature added as part of Tjasink's Masters degree. Also in development is a Just-In-Time compiler for it named *cavalry* which is written in Java. *Kissme* is still a work in progress.

2.2.2 JOP Basics

The JOP forgoes all of the upper levels of the JVM, running it right down to the basics. This optimized JVM must have an external compiler and editor to create a .class file. It doesn't have a dynamic Class Loader so the .class file, for this implementation at least, must be placed in a predefined static location, ie: the ROM, where the JOP can access it.

Once the JOP has the bytecode, it acts in a similar manner to the JVM. It has the equivalent of an Execution Engine which translates the bytecode to hardware specific code and runs the commands necessary to execute this code before translating the results back. All the code for this resides in the Core of the JOP (Section 3.2). The results are saved and would require an external application to translate them to a human readable or usable format.

Future work for this project will be discussed in Chapter 7. External applications mentioned here could be created and tied in with the work done on the JOP so far. Making specific applications for this would remove unnecessary reliance on external software.

To optimize further, the Support Code has also been removed, or rather, been moved closer to the Execution Engine. The JOP has been stripped to its bare bones essentials discarding superfluous methods such as the ability to manipulate multiple threaded environments.

The JOP is already so well optimized in its internals that it was decided to abolish the Garbage Collection Heap. Due to limitations of the instruction set, and the existence memory reclamation code in each segment of the JOP, the Garbage Collection Heap became redundant and thus provided an opportunity to free up much needed microcontroller space.

The preceding optimisations shrink the already small JVM to such an extent that it will comfortably operate on an FPGA microprocessor having come from an ample PC environment. The RISC architecture on the Virtex II

boards makes programming them an arduous task. The reduced instructions means that the microprocessor is now only programmable with a limited set of languages. These languages are only one or two steps removed from actual machine code and contain very simple instructions. Having a JVM implemented on these chips will provide more functionality and usability than that available with RISC architecture.

Chapter 3

JOP Design

3.1 Breaking Down The Code

The original code was downloaded directly from Michael Schoeberl's website[14]. It had to be split up, broken down and understood. The line count of the original code plus libraries from Altera[2] and the IEEE[5] totals 11,130 lines. The final line count totalled a mere 4,575 lines, much less than half of the original count.

The original code was analysed and described using pseudo code to better understand the application flow and module interactions and dependencies. This translation afforded a simplified overview of the entire code structure allowing easy manipulation of the comprising elements. Areas which were at this time incomprehensible could then be identified, isolated and studied autonomously.

The next logical step undertaken was a more specific overview of the program as a whole. This in turn lead to the view of components shown in *Figure 3.1* as well as a detailed observation of how they interact.

The following sections describe the original design and code implementation. Subsequent alterations and additional components which were a direct result of this project will be detailed in Chapter 4.

3.2 The Core

The Core component encapsulates much of the memory, Stack and ALU functions. It gives the JVM much of its functionality and can also be split into sections for better understanding.

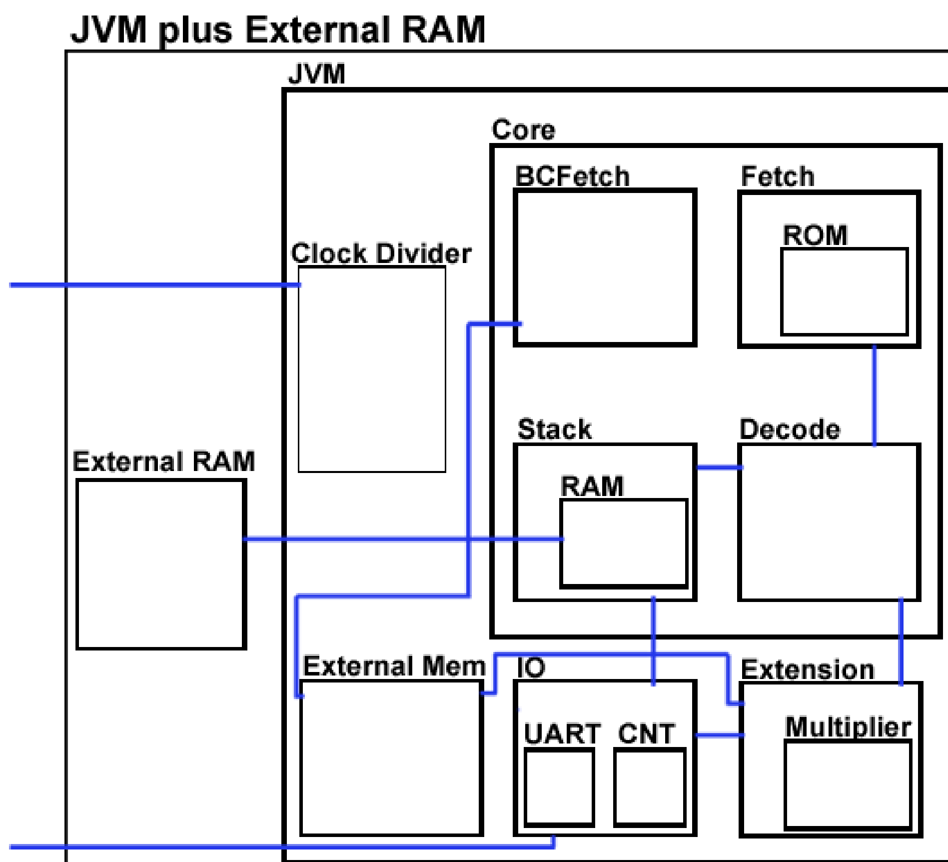

Figure 3.1: Components of the JVM

3.2.1 Fetch component

The Fetch component works with the ROM, feeding bytecode instructions from the ROM directly to the Decoder where they will be translated to JVM addresses. Bytecode was explained in Section 2.2.1

Data in the ROM is 10bits long and contains bytecode. As mentioned previously, bytecode is created when a Java program is compiled and is an intermediate layer of code between source and machine code[9]. It is stored into the ROM and is accessed by Fetch. From Fetch it is sent, via the core, to the Decoder where the bytecode will be split into instructions.

A segment of the ROM code is located in Appendix A. The entire code, too long to include here, is located on the disk provided with this report.

3.2.2 Bytecode Fetch component

In the Bytecode Fetch part of the JVM, *jump* and *branch* instructions are decoded and flags are set so that they can be recognised and executed when they reach the stack. The signals *busy* and *branch* are then sent back to Fetch to stop other instructions getting through, in the case of an interrupt, or to change the memory address of the next instruction, in the case of a branch.

3.2.3 Decoder

Here the bytecode is input from Fetch and translated to a JVM address. The mappings for the instructions are shown in *Table 3.1*.

These instructions show the importance of the Stack. Explanations of all the above instructions are available in *timing.pdf* on the JOP[14] website. *Table 3.2* has been reproduced here based directly on Martin Schoeberl's work and explains the registers and memory areas used by the optimized JVM. Additional information is available from *timing.pdf* in which pages 11 to 33 provide a full explanation of all of the above instructions.

3.2.4 Stack

JVMs are stack based. The stack is split into three main sections: one section holds local variables, another exists as the execution environment and the last holds the operand stack. The execution environment maintains the current operations of the stack and the operand stack is a workspace for bytecode instructions and parameters[15].

The Stack component does not just implement the typical push and pop instructions. This component is an overlay on the three sections of the stack mentioned above and as such it can also perform all the logic and arithmetic methods for the system, working on the JVM addresses sent to it by the decoder. The original Stack held a 32bit RAM, alterations had to made to this and will be detailed in the next chapter.

Most of the RAM is filled with the hexadecimal value &H12345678*. However, there are some constants sent in. The values of the RAM file are located in Appendix B.

*Throughout this report, whenever &H is used it is prefixing a hexadecimal value

Bytecode (in hexadecimal)	Instruction	Bytecode (in hexadecimal)	Instruction
00	pop	2X/3X	stm
01	and	4X/5X	bz
02	or	6X/7X	bnz
03	xor	80	nop
04	add	82	jbr
05	sub	aX/bX	ldm
08	stioa	cX/dX	ldi
09	stiod	e1	ldiod
0a	stmra	e2	ldmrd
0b	stmwa	e3	ldmbsy
0c	stmwd	e5	ldmul
0d	stopa	e8	ld0
0e	stopb	e9	ld1
10	st0	ea	ld2
11	st1	eb	ld3
12	st2	ed	ld
13	st3	f0	ldsp
15	st	f1	ldvp
18	stvp	f2	ldjpc
19	stjpc	f4	ld_opd_8u
1a	not used	f5	lp_opd_8s
1b	stsp	f6	ld_opd_16u
1c	ushr	f7	ld_opd_16s
1d	shl	f8	dup
1e	shr		

Table 3.1: Mapping of bytecode to JVM address

3.3 Extension

Extension provides an interface between External Memory, IO, a Multiplier and the Core, mapping each to the other. Depending on signals coming from the Core it schedules reading and writing from each of these components. The address mapping is shown in *Table 3.3*, the address coming from the Core where it was encoded with the bytecode coming from the RAM and translated in the Decoder before being passed on to this external component. Some addresses are duplicated in the table because they will be sending

Name	Description
A	Top of the stack
B	The element one below the top of stack
stack[]	The stack buffer for the rest of the stack
sp	The stack pointer for the stack buffer
vp	The variable pointer. Points to the first local variable in the stack buffer
pc	Microcode program counter
offtbl	Table for branch offsets
jpc	Program counter for the Java bytecode
opd	8 bit operand from the bytecode fetch unit
ioar	Address register of the IO subsystem
memrda	Read address register of the memory subsystem
memwra	Write address register of the memory subsystem
memrdd	Read data register of the memory subsystem
memwrd	Write data register of the memory subsystem
mula, mulb	Operands of the hardware multiplier
mulr	Result register of the hardware multiplier
member	Bytecode address and length register of the memory subsystem

Table 3.2: JOP hardware registers and memory areas

instructions to multiple components.

3.3.1 Multiplier

The Multiplier implemented in Extension uses the Booth Algorithm. Each computation takes as many clock cycles as there are bits in the multiplicands. Here the multiplicands are all 32bits long so every multiplication takes 32 clock cycles.

The algorithm takes three bits at a time and, depending on the pattern of the bits, performs the action indicated in *Table 3.4*[11]. The string mentioned in the Operations column is *A*, and these operations are performed on *B*. In this way *A* and *B's* bits are added up accordingly to complete the multiplication.

Address	Operation
0	IO address
1	IO read/write
2	Store memory read address
2	Load memory read data
3	Store memory write address
4	Store memory write data
5	Load multiply result
5	Store multiplicand A
6	Store multiplicand B and start multiply
7	Store start bytecode load (or cache)
7	Load read new pc base (for cache version)

Table 3.3: Address mapping for Extension

	Bit		Operation	M is multiplied by
2^1	2^0	2^{-1}		
B_{i+1}	B_i	B_{i-1}		
0	0	0	add zero (no string)	+0
0	0	1	add multipleic (end of string)	+A
0	1	0	add multiplic. (a string)	+A
0	1	1	add twice the mul. (end of string)	+2A
1	0	0	sub twice the m. (beg. of string)	-2A
1	0	1	sub the mul (-2A and +A)	-A
1	1	0	sub the mul (beg. of string)	-A
1	1	1	sub zero (center of string)	0

Table 3.4: Operations of a Booth Multiplier

3.4 IO components

IO controls any input and output data ports to the JVM, as one would expect. It also overlays a counter for clock signals and an interface with a UART[†]. As such, this component plays a pivotal part in the code. It manages how the UART and Counter interact, not only with each other, but with the rest of the system. To do this it, of course, employs address mapping and internal signals. The mapping is shown in *Table 3.5*.

[†]Universal Asynchronous Receiver-Transmitter

Address	Operation
0	Read clock counter, write interrupt enable
1	Read 1 MHz counter, write us counter and interrupt acknowledge
2	Write generates interrupt
3	Write watchdog port
4	UART (download), transmit data register
5	UART (download), read data register

Table 3.5: Address mapping for IO

3.4.1 Universal Asynchronous Receiver-Transmitter

The focal point of the IO exists in the UART component. It operates on a stack based FIFO[‡] algorithm, using variable clock signals to control the receiving and sending of data. The receive clock changes every 8 system clock cycles while the transmit clock only alters every 16 clock cycles, taking twice as long.

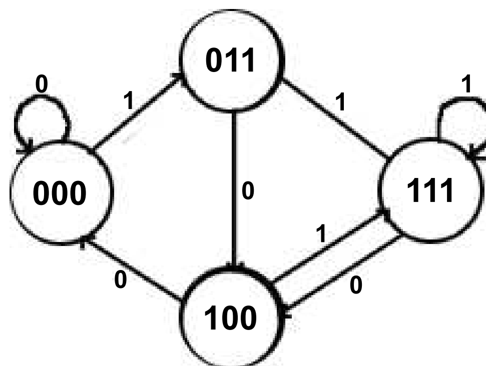

Figure 3.2: Receive and Transmit Buffers

Shown in *Figure 3.2* are the signal changes in the *ncts*[§] and the *nrts*[¶] buffers. These signals control interaction with the FIFO, specifying how it should be operating at any given time. They implement a simple FSM[‖], cycling through the same four states over and over.

[‡]First In First Out
[§]clear to send, ie: another UART is ready to receive data
[¶]request to send, ie: this UART's FIFO is cleared and able to accept data
[‖]Finite State Machine

The FIFO portrayed is incredibly simplistic, using a basic stack structure. If a write signal is input it saves the data coming in and outputs it until the stack is emptied. The FIFO will continue to take in data but will only output the first value entered with the write signal unless the stack has been emptied.

If a read signal is sent to the FIFO, the stack is emptied and the output cleared. This makes the FIFO ready for new input which was read in previously.

3.4.2 Clock Counter

The Clock Counter entity controls the interrupt handling and sets the *watchdog* bit if necessary. It generates two clock signals; a clock counter and one called *us_counter* and has a tri-state output:

1. If the signal *addr* is zero, the clock counter is output.

2. If the signal *addr* is one, the *us_counter* is output.

3. Otherwise, the output is undefined and set to 'Z'.

The clock counter output simply counts clock signals, as its name would imply. The *us_counter*, on the other hand, increments every 99 clock signals, effectively creating a 1 MHz counter.

As well as the timers, the Clock Counter generates interrupts. Using the address mapping inherited from the overlying IO and shown in *Table 3.5*, the Clock Counter sets the variables which set the interrupts.

3.5 External Memory

The External Memory component interacts with the Cyclone board. It provides a link to the external RAM and ROM on the board as well as to an external NAND flash chip. The RAM contains 32bit words while the ROM contains 8bit ones. They are referenced using the memory mapping shown in *Table 3.6*.

Memory Address	Memory Unit
000000 - x7ffff	External RAM
080000 - xfffff	External ROM
100000 - xfffff	External NAND flash

Table 3.6: Memory mapping for External Memory

Chapter 4

JOP Implementation

Many changes had to be made to the original code to get it operating correctly on the Virtex II board. One of the biggest problems encountered in the course of this project was that the code was originally written for Altera[2]. College resources employed in this endeavour meant that the entire program had to be ported over to Xilinx[17], a rival of Altera. Altera and Xilinx provide very different libraries, methods and editors for programming VHDL. Also, due to the removal of some unportable components or size issues, new components had to be introduced to the JOP. All of this had to be taken into consideration and every component of the JOP had to be altered accordingly.

4.1 Generic Parameters

Some built-in features of Altera's editors were not portable to Xilinx. These had to be replaced with newly coded equivalents. One such feature was generic parameters. Generic parameters were used in almost every part of the JOP. They are dynamic parameters. These parameters are often defaulted but can also be reassigned throughout the code at either component level or somewhere hierarchically higher up. They act as globals and are accessible throughout the application, including in the port section. This enables dynamic inputs and outputs which can be of any size, as determined by the generic parameter.

Unfortunately, Xilinx does not allow generic parameters[10]. This meant that every generic parameter had to be removed throughout the code. Every reference to a generic parameter had to be replaced with the value that generic would hold in this version of the JOP. Therefore, wherever a generic occurred, the entire hierarchy had to be traversed to find the original value for it, or to determine that it was loaded with its default value. This was

long and time consuming but, after it was complete, many of the components ran on the Xilinx compiler and just needed to be tested (Chapter 6).

4.2 Altera Library Files

Some Altera library files were used in the original code. These had to be broken down and understood before they could be replaced. A lot of the code was very obfuscated, being intended to be readable by machines rather than humans and this slowed down the process.

It was eventually discovered that the main libraries employed by the original code were for reading and writing external files. The memory units made the most extensive use of these libraries. ROM and RAM programs and data were saved in external files of type .mif*. These files program memory units in Altera but not in Xilinx. In order to port over to Xilinx, these files had to be removed. This meant re-writing all of the memory units in the JOP. Using the .mif files as a basis, virtual memory units were created with variables which held arrays of memory for the JOP, the same values as before but accessed in a different manner.

Segments of the ROM and the full RAM .mif files are included in the Appendices A and B respectively.

4.3 Other Methods

Various disparate code segments would not port directly to Xilinx. These were identified and rewritten. Many were not discovered until the full JOP was running during the testing phase. The ramifications of each change could then be better ascertained and incorporated into the re-work. The 'knock-on' effect of these changes were not significant and will not be detailed here.

4.4 Core

Many parts of the Core were drastically altered to enable them to run on the designated software and hardware platforms. The necessary Core changes resulted from not only editor, compiler and synthesiser incompatibilities but also from the size constraints imposed by the hardware.

*Memory Initialisation File

4.4.1 BCFetch

In the original code, the bytecode with *jump* and *branch* instructions was fetched from External Memory. Due to size differences between the original and final hardware platforms' memory units and the discrepancies mentioned in Section 4.2, these functionalities had to be removed. The fact that the bytecode could not be successfully reverse engineered also contributed to the discontinuation of these instructions. There is room for this memory unit to be put back in place (Chapter 7). To facilitate this the *jump* and *branch* instructions have been defaulted to zero and the the associated interrupts disabled. As previously mentioned the memory module can be refitted on the board and indeed this would have to be the case before the aforementioned instructions set were to be re-enabled. It was deemed unnecessary to do so at this time and remains outside the boundaries of this project.

4.4.2 Stack

The Stack implemented in the original code used a 32bit RAM. Unfortunately, due to space constraints on the board, this had to be downsized. It is now an 8bit RAM. To avoid data loss it is treated like its 32bit counterpart. By manipulating the clock, four consecutive 8bit memory addresses are accessed and appended to be output as a 32bit value. If the RAM is about to be written, the memory is split and fed into these addresses. The clock pulse feeding to the RAM is four times faster than the system clock to allow for this.

With this altered RAM and a Shift Register, the Stack still performs all the operations sent to it by the Decoder.

4.5 External Memory

Due to the internal RAM already being made external (Section 3.2.4) there was a lack of available memory on the board. Therefore it was decided that the External Memory unit was no longer needed. Its memory units are not needed and so were removed, freeing up much needed resources. Its only current use is to output control signals to the JVM to ensure smooth running.

4.6 Short RAM

This section should further explain the RAM mentioned in Section 3.2.4. The original code, as mentioned previously in this chapter, used 32bit values. This proved too large to fit on the board so allowances had to be made for it.

Originally it was thought that moving this RAM to the external chips on the board would solve the problem. This meant structurally changing the code and re-writing the internal RAM so that it acted as a conduit to the outer chip. In order to retain the same usability in memory accesses, it was decided to create a new clock pulse (Section 4.7) which was four times quicker than the system clock. This allows four addresses to be accessed per system clock pulse.

An Addition component was created to manage the addresses being accessed because the Booth Multiplier (Section 3.3.1) would have taken too long. The addresses fed out to the on-board RAM, through the conduit component are the same as the original ones. However, because the data values are now 8bits long instead of 32bits, the addresses must be converted accordingly. This means multiplying them by four and incrementing them by one each time the quicker clock pulses. This will be discussed in greater detail in Section 6.9.

To properly test the RAM conduit, and show it working during the demonstration, a virtual prototype RAM was created. This was another VHDL module which was programmed as an 8bit RAM containing values which the original RAM had held. It was intended to represent the on-board RAM and synthesise the full working JOP on the host computer. A top level component was also written which would connect the JOP code to this RAM.

During testing it was observed that the JOP code plus this new virtual RAM, was small enough to fit on the FPGA. It contained all the data of the original 32bit RAM but was structured and accessed differently. On further testing it was obvious that the JVM could now exist using either the external on-board RAM or this new 8bit RAM. The new RAM is located outside the JOP but is internal to the program so will be loaded onto the FPGA.

The version of code which is implemented dictates which of the two methods of RAM access is utilised. Using the JOP top level file, written specifically for this reason (Section 4.8), the new 8bit RAM is included. If the JOP module one step lower in the hierarchy is used then the virtual RAM is excluded and the on-board RAM must be used.

At this point a RAM writer was also created. When loaded into the FPGA this file takes the original 32bit values and breaks them down to 8bit pieces, feeding each piece into an adjacent address. When the write

is finished, the program will flash the LEDs available on the board in a "1010101" pattern. The lower level JVM can be loaded onto the FPGA and will access the external on-board RAM in exactly the same manner as before with no differences in the output.

4.7 Clock Divider

The Clock Divider does exactly what it says it does, it divides up the clock. It's a very simple method of creating different clock pulses. The incoming clock to the FPGA is fed directly to this module. Once there, the clock pulse is converted to a 32bit signal. By taking say the 24th bit of this signal, one can effectively slow down the clock to 2^{24} or 16,777,216 times slower than its original speed of 25MHz. This means that the clock is now running at 14x10^{-7}MHz. Look at *Table 4.1* to clarify the matter. The table shows zero to seven being counted in binary. If you look at the first bit, it changes, one to zero and vice versa, each time the number increments. However, the second bit only changes with every two incrementations, or 2^1 times. In turn, the third bit changes every 2^2 or four incrementations. From this it should be obvious that to get a slower clock cycle one must simply take a more significant bit since the pace decreases by a factor of two with each bitwise shift toward significance.

Bit 2	Bit 1	Bit 0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Table 4.1: Binary bits of count zero to seven

For the demonstration, the clock was slowed considerably so that outputs could easily be displayed and more readily observed on the LEDs. The slowest clock available on the Virtex II boards was 25 MHz. By taking the 24th bit this was slowed to a pace the human eye could follow. The true purpose of this Clock Divider however is to operate the RAM properly. This component will output bit zero and bit two of the clock, thereby giving the RAM a direct

clock signal and the rest of the JVM a signal four times slower. The system could be set to operate using only the 24th bit but it would prove extremely inefficient.

An adaption of this divider could help students in future who are doing experiments on the board.

4.8 Top Level of Design

As mentioned in Section 3.2.4, there are two possible top layers. One links the Clock Divider, the External Memory unit, the IO, Extension and Core components together and provides the inputs and outputs to the JVM, including those applicable to an external RAM. The second, built upon the first layer, links the virtual 8bit RAM to the JVM code and only deals with inputs and outputs integral to the working of the JVM itself.

4.9 Pipeline Architecture

All of the code put together runs in a pipelined architecture. Shown in *Table 4.2* is a 5-stage pipeline. Instruction Fetch(IF) is the first stage, instructions are fetched from memory. After this comes Instruction Decode(ID), the decode stage where the instructions are prepared for processing. The third stage is EXecute(EX), where the instructions are carried out. The fourth and fifth stages, MEMory access(MEM) and Write Back(WB), can differ from system to system. The memory access stage writes any results from the execution into memory. During the entire operation, some variables may have been altered which need to return to their original values, the write back stage takes care of this.

IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Table 4.2: 5-stage pipeline

The pipeline works because each set of operations will be on the pipeline at a known amount of time. Instructions are fetched in the first stage. When an operation has fetched its instructions it progresses to the second stage

in the pipeline. This leaves the next operation in the queue free to fetch its instructions and in turn enter the pipeline. When the first has completed the second stage, the second finds it vacated and progresses to this stage. This progression continues while there are still waiting operations. This sequential loading of the pipeline ensures that several operations can be running concurrently and, because none of them are accessing the same resources at the same time, incur no conflict errors. This greatly speeds up the running of the system.

The JOP works on a similar pipeline, although it lacks a WB stage and has a few alterations to allow for the differences with other architectures. The stages are shown in *Figure 4.1*, taken from *Design Rationale of a Processor Architecture for Predictable Real-Time Execution of Java Programs*[14]. Each stage implements the previously mentioned components of the JOP. They all have to be carefully timed and controlled to fit into the pipeline as intended. In the first stage, bytecode instructions are fetched, these are translated to addresses in microcode. Any bytecode branches or interrupts are also decoded and executed here. In the second stage, microcode is fetched. This microcode comprises the actual instructions of the JOP itself, rather than Java bytecode. Any branches in the microcode are taken here. The third stage decodes the instructions. It also generates a stack address in the RAM. It is possible to generate addresses for a following instruction here and thus speed up the running processes. The memory access and write back stages shown in *Table 4.2* are no longer necessary when the execution stage is reached. Operations in the execute stage are performed on the TOS[†] and TOS-1 registers. Data between these registers and the Stack are moved, ie: filled or spilled, during this stage. This makes a specific stage for write back and memory access defunct.

[†]Top Of Stack

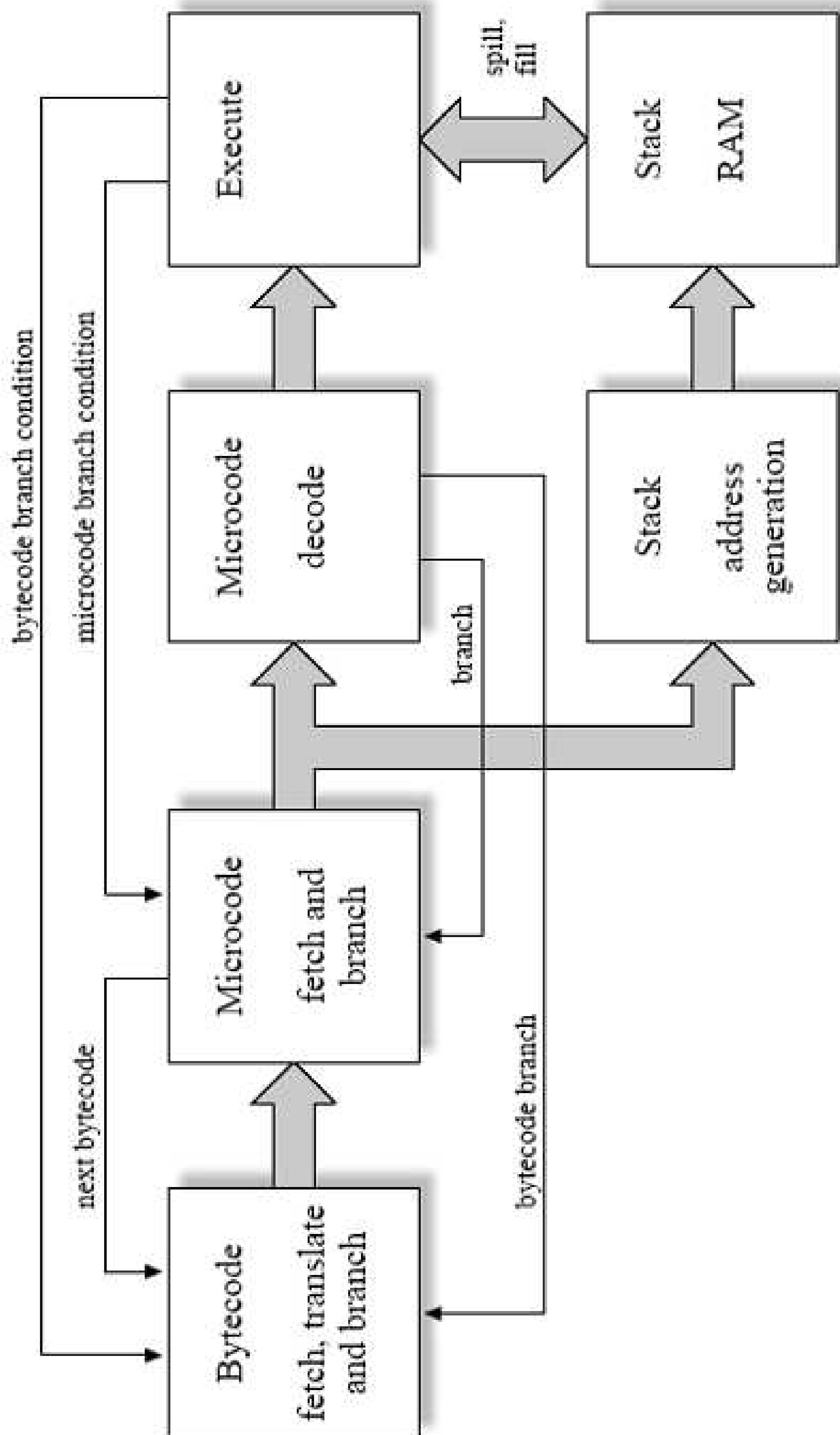

Figure 4.1: JOP Datapath

Chapter 5

Hardware Analysis

5.1 Hardware Description

There were two boards used in Ross' project[3] (Chapter 2). Both were Virtex II prototype boards[4]. These boards allow a designer to mount additional chips and wire new circuits. The prototype model encourages rapid development, flexibility and real world experimentation without incurring the monetary and time cost of once off PCB* fabrication.

Both Virtex II boards are wired almost identically except that one has a CPLD† chip mounted and the other doesn't. During the course of this project the prototype board without the CPLD was used. Any references to a 'board' may be assumed to reference the Virtex II prototype board under analysis unless otherwise specified.

The board comes with some chips already in place. Among these it has a PROM‡ for programming purposes. Also connected are two FPGA's; a Service FPGA and a DUT§ FPGA. In the course of this report, whenever reference is made to the FPGA, it is the DUT FPGA which is being discussed.

The DUT can be configured, or programmed. It is easily changed and provides the real platform upon which this project was built. There are several ways to program it, one method involves directly downloading the program in Slave Serial mode. To do this a Parallel Cable IV was connected to the board, through which the program could be downloaded using Xilinx's iMPACT tool. This method was used extensively in the course of this project. Another way involves downloading a program from the PROM which would have been programmed earlier. The board used here had a preprogrammed

*Printed Circuit Board

†Computer Programmable Logic Device

‡Programmable Read Only Memory

§Device Under Test

PROM which lit the LEDs, showing quite primitively the correct operation of the entire board.

Shown in *Figure 5.1* is the Virtex II board used in this project. The components just mentioned are visible as well as a few which will be referenced later in this chapter. On the top left is a vertical line of pins, these are the Configuration Port User PROM and FPGA header pins. Beside these pins is a red box with white dipswitches, these are the clock enable switches. Below the dipswitches are four grey switches which hold, going downwards, the Clock Frequency changer, the Configuration Mode switch, the JTAG mode switch and the Chip Select. To the right of these switches is the development area. Placed here are the RAM, ROM and max32 chips. Above the development area are two chips. The one to the left is the PROM. It is placed in a holder from which it can easily be removed to prevent accidental bending of the contact pins. To the right is the Service FPGA. Further over to the right, located amongst several pin connectors, is the DUT FPGA, which is also located in a holder. The pin connectors can be wired to any device in the development area. Below the FPGA, near the bottom of the board, are eight LEDs, which can be used for testing sequences, and the Reset and Program buttons. To their right are two more LEDs, these representing Done and Init. Init is lit when a program is being loaded, and Done is lit when the program is loaded.

5.1.1 Configuration Modes

The Board has several different modes of operation built in. The only two configuration modes explored in the course of this project were JTAG[¶], which is a testing mode, and Slave Serial mode. By using switches already on the board any of eight separate configuration modes could be in use, as shown in *Table 5.1.* The definition of PROM and Upstream in the table represent what method of input is going to be used, i.e.: whether the new program will come from the PROM or from the Upstream Configuration Interface connector located at the side of the board.

5.1.2 JTAG Mode

JTAG mode deserves some mention as it has its own subsections. JTAG is generally accepted as a testing mode which, on a prototype board, gives it a primary position.

[¶]Joint Test Action Group

Figure 5.1: Virtex II Prototype and Demonstration Boards

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

Table 5.1: Configuration Mode

As well as the configuration switch on the board, there is also a JTAG switch. This comes into use when the config switch is set to 5. When this point is reached, the testing mode can be selected using the JTAG switch located below the config switch on the board. This switch allows testing of the FPGA and the PROM in different circuits according to the modes shown in *Table 5.2*. It allows loading of programs directly to the FPGA or to the PROM, as well as sending the program from one to another, as shown in modes 2 and 3 of *Table 5.2*. If the program is saved to the PROM and the board switched to JTAG Mode 3, the FPGA will be loaded with the saved program in the PROM. This is an easy way to program the FPGA, which loses its program each time the board is turned off.

Switch Position	JTAG Mode
0	FPGA
1	PROM
2	FPGA > PROM
3	PROM > FPGA
4	Feed Through

Table 5.2: JTAG Mode

5.1.3 Slave Serial Mode

The Slave Serial mode is used for this project due to problems discussed in Section 5.4. Using a Parallel Cable IV, with wire mappings shown in *Table 5.3*, the board is connected using the Configuration Port User PROM and FPGA header to a computer. The cables on the Parallel Cable IV correspond directly to those of the Configuration Port. The computer holds the code for the JVM. The code is downloaded to the board and programming the FPGA only takes a few moments.

5.2 FPGAs in Detail

Producing project specific hardware chips would prove prohibitively expensive. As with the boards discussed earlier, inbuilt flexibility is the best solution to keeping costs to a minimum. For this reason on-site or *'field'* programmable chips have become the industry standard for most PCB devices. From this comes the FPGA, or Field Programmable Gate Array.

Parallel Cable	Board Pins
VREF	VCC3
GND	GND
CCLK	CCLK
DONE	DONE
DIN	D0
PROG	PROG
INIT	INIT

Table 5.3: Parallel Cable connections

Re-programmable devices have been increasing in popularity over the years. By now a lot of circuits have dynamic rather than static chips on them because they allow more freedom to the designer of the circuit. The FPGA is just one more step in that direction, it can be reprogrammed whenever needed and is excellent for testing purposes. Using a static chip in such circumstances would be wasteful especially if the logic did not work first time.

The internal circuitry of the FPGA allows it to be reconfigurable by downloading new logic into its memory so that its internal logic reconfigures itself to allow for the new functionality. An FPGA can be thought of as an IC[‖] with logic gates connected with the equivalent of fuses. Originally, the fuses are intact; programming the device involves blowing the appropriate fuses to make the required circuit. The FPGA returns to its original state on power off, due to the FPGA's circuit functions being saved on SRAM[**]. Using RAM memory makes the FPGA easily and quickly configurable, however it also makes it volatile. As a result, the FPGA must be reprogrammed every time it is powered off[8].

5.3 Putting the JVM onto the FPGA

This project was completed with various tools, including many belonging to Xilinx. Using Project Navigator, the JVM code was converted from Altera to Xilinx (Section 4). Extensive testing followed using ModelSim tools to testbench and check outputs and waveform signals (Chapter 6). Following the simulation testing part, the JVM had to be downloaded to the FPGA

[‖]Integrated Circuit

[**]Static RAM

for on-board testing. This led to many unforeseen problems which will all be covered in Section 5.4.

When converting the code to a format understandable by the FPGA, size remains an important consideration. The resultant .bit file must physically fit in the available memory of the FPGA. Project Navigator provides tools for this occurrence. First, a User Constraint file has to be created. This file has a .ucf extension and is viewable as a text file or by opening PACE. PACE is an editor with which the user can graphically assign the pins of the FPGA to the inputs and outputs of the program.

Once the .ucf file has been created, Project Navigator can implement the design. This process involves mapping out the pins and translating from code to hardware format. A place and route process also runs at this point which decides where each section of code will be implemented in the FPGA, i.e.: how the FPGA's circuits will be redefined to provide the functionality described in the code.

Project Navigator is now finally able to generate a programming file, the least time consuming process of all. This programming file will be used to reconfigure the FPGA.

Once this programming file has been generated, saved as a .bit[††] file, iMPACT can be opened. iMPACT is a tool provided by Xilinx. It interacts between the host computer and the hardware board to program the latter. A Parallel Cable IV connects the board to the host computer (Section 5.1.3). The Parallel Cable IV's cables correspond directly to the Configuration Port User FPGA and PROM header on the board, as shown in *Table 5.3.* Once these have been connected up, and the board plugged in and turned on, the config switch should be turned to 4 to allow for Slave Serial programming. Next, turning to the iMPACT window, and choosing Slave Serial mode, a device can be added using the .bit file created earlier. By right clicking the device and choosing *'program'* the bitstream file is downloaded to the FPGA, programming it with the original .vhdl code.

5.4 Additional Problems

After the board was programmed several fundamental problems were discovered. Through the synthesis and translate processes it was discovered that the requirements of the JVM exceeded the resources of the Virtex II boards. It was simply too large to fit on the board. As discussed extensively in Section 3.2.4 and Section 4.6 in Chapter 3, removal of the RAM and inclusion of a smaller one eventually solved this problem.

[††]bitstream

When the code was the right size and format for downloading, a problem with iMPACT was discovered. It was originally decided to use the JTAG mode for downloading and testing the JVM on the FPGA. Unfortunately iMPACT, on several computers, refused to work in JTAG mode. To resolve this a change to Slave Serial mode had to be made. JTAG mode is an accepted testing mode but, due to circumstances, all testing had to occur in Slave Serial mode. This limited the application's functionality. Specifically the option to save the .bit file to the PROM, which would have enabled the JVM to be loaded whenever the board was turned on, was unavailable. Instead, the bitstream file had to be downloaded every time the board was rebooted. Although not prohibitive, this did have an impact on the time taken for testing the code.

As well as this, the clock speed of the board was a major problem. When the code was first placed on the board it was impossible to tell whether or not it was working correctly because the clock speed was too fast. LEDs connected to the FPGA lit dimly indicating that there were outputs, but fully lit, unlit or flashing LEDs were indistinguishable from each other. This problem was solved by the addition of the Clock Divider component (Section 4.7), which slowed down the clock to speeds the human eye could follow.

A final additional problem with the board arised when testing of the code required a Reset signal. The Reset button on the board does not work. The Reset on the second Virtex II board was also testing and found to be malfunctioning. This did not threaten the project as the top level of the code simulated its own Reset signal. However, it did hinder testing of lower levels of the code on the board.

Chapter 6

Evaluation & Results

Much of the testing of the JOP took place on a PC, using Project Navigator and ModelSim tools. As mentioned in Section 3.1, the code was broken up into more manageable parts. Working from the bottom level up, each piece of code had to be changed and then tested extensively by comparing waveformations to the code. The code was tested as it was broken up and will be described as such, thus creating building blocks for the final testing on the board.

6.1 Stack Testing

The Stack was the first part of the code which was ready for testing. The Shift Register and RAM conduit were independently tested prior to a combined investigation into the overall functionality of the Stack. Shown in *Figure 6.1* are the waveforms produced by the stack code running. The values shown are in hexadecimal for easier understanding.

The red lines at the far left of the waveform represent unassigned values. Nothing in this system is set until *reset* has toggled low. *Reset* is located at the top left of the diagram. It is set to active-high, so as soon as it changes to 0, the Stack can begin functioning.

The Stack's waveforms are quite complex. It implements an ALU* for the JVM. Most of the waveforms in *Figure 6.1* are inputs, calculated to test for specific outputs. The last six waveforms represent the outputs of the Stack, *aout* and *bout* being the most important. The first combination of input signals loads the input *din* into both *aout* and *bout*. This can be seen at the start of the waveform where the outputs are the same as the input. The

*Arithmetic Logic Unit

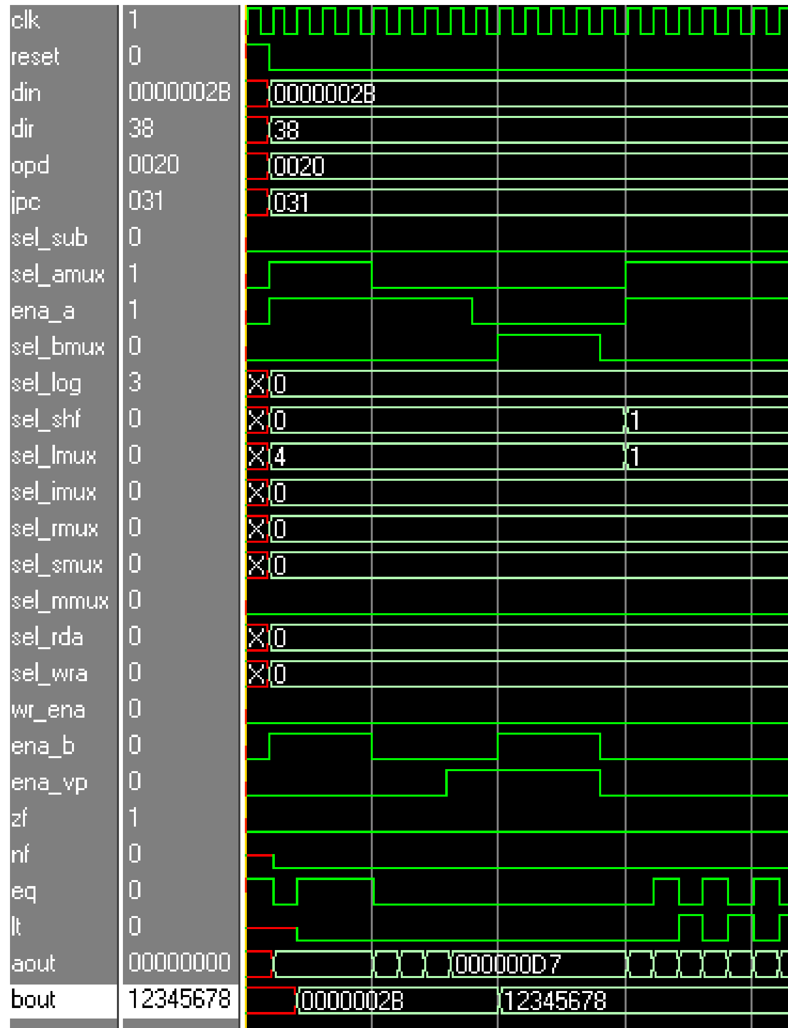

Figure 6.1: Stack Testbench Waveform

next change in *aout* is caused by *sel_amux*† going to zero, this causes *aout* to become the sum of *aout* and *bout* which is &H56. All that is visible in printed form is the toggle in *aout* at that point. For three further system clock pulses, *aout* continues to be set to the sum of *aout* and *bout* thereby incrementing by &H2B each time and resulting in *aout* holding &HD7. Before the next clock pulse, the signals are changed once more, resulting in *bout* being loaded with the value &H12345678 from RAM. This value remains constant for the rest of the waveform. Turning back to *aout*, the selects, located in the center of the waveform, are now changed. *Sel_shf* and *sel_lmux* both change to one,

†sel_amux is the eight waveform

this will force a shift to the left of *aout's* value. It is here that some of the other outputs begin to toggle. *Aout* and *bout* are, at every second clock pulse, equal to each other, so the *eq* waveform toggles to one and back, true and false, to represent this. Take note that the other time eq was set to one was at the start of the waveform when *aout* and *bout* were first both unassigned and then both filled from *din*.

6.2 Fetch Testing

Fetch implements the ROM and the offset table for bytecode operands. Fetch's waveform is much easier to follow than that of the Stack. The main output is *dout*, situated at the bottom of the waveform. It's clear that this is constantly changing. It is actually accepting and outputting ROM data on every clock pulse. It runs through the program located on the ROM, fetching and outputting the operands required by the system.

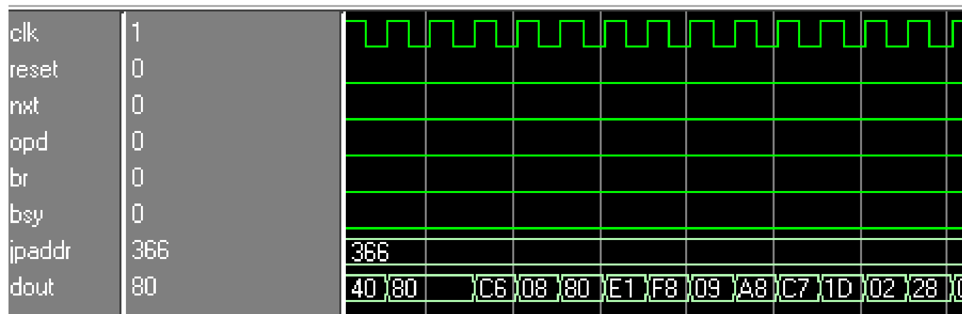

Figure 6.2: Fetch Testbench Waveform

6.3 BCFetch & Decode Testing

BCFetch and Decode were tested together to save time. Unlike the other test-benches, the outputs *jpc_out* and *jbc_addr* of BCFetch are mixed in with the inputs while the rest are located, as usual, at the bottom of the waveforms. *Jpc_out* is set to zero until *jopdfetch* goes high, after which *jpc_out* increments by one every clock cycle. One can observe that, due to *jopdfetch* going high, *jbc_addr* also begins to increment every clock cycle. In this case, however, this is caused by *jbc_addr* being equal to *jpc_out* plus one on each clock cycle. Therefore, when *jpc_out* increments, so too does *jbc_addr*. *Jpaddr* is the address of the next branch command. When bytecode is fed through the jump table *jpaddr* is the resulting address. *Opd* also changes when *jopdfetch* goes

high. While *jopdfetch* is low, the first two bytes of *opd* are set to zero, while the last two are set to one of the inputs: *jbc_data*. When *jopdfetch* goes high, its first two hex numbers are now filled with the last two bytes. It should be clear from this that *jopdfetch*, and also *jfetch*, which was not shown here, are control signals for BCFetch.

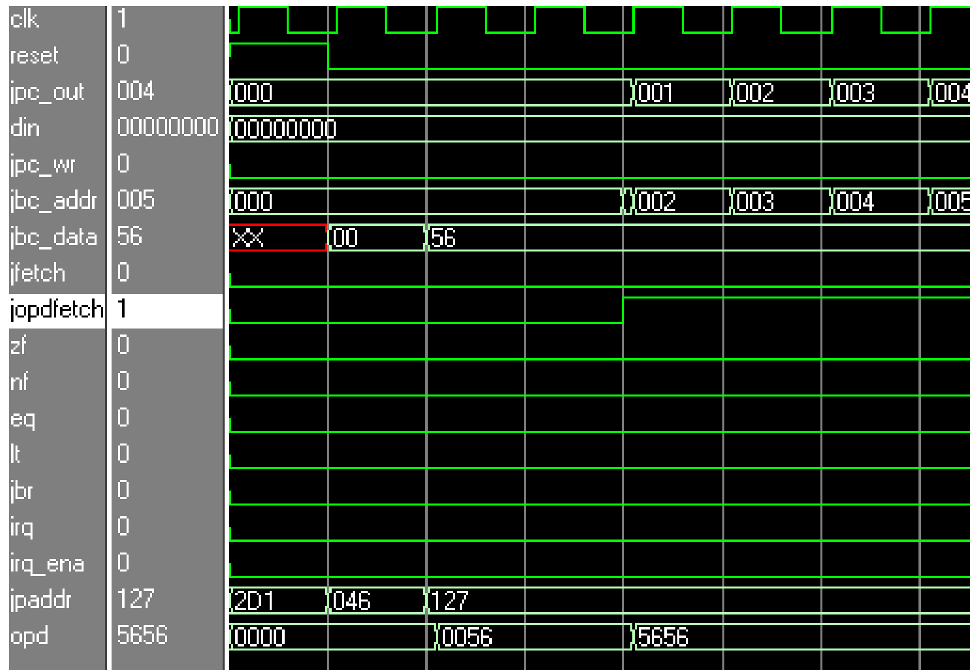

Figure 6.3: BCFetch Testbench Waveform

The Decoder has a multitude of inputs and outputs. It receives instructions from Fetch and breaks them down into signals for the Stack. These are the select and enable signals visible in the waveform. *Table 3.1* shows the mapping from the input instruction to an actual operation. The output signals from the Decoder will ensure that the correct operations are performed elsewhere in the system.

6.4 Core Testing

The Core interlaces the previous components. Prior to putting it all together, these components needed to be tested. When they were proven to be working correctly, the Core itself had to be tested. This proved to be overly complex so a basic test was devised. The waveform for this test is shown in *Figure 6.5*. To facilitate the testing, the program from the ROM is run on the Core. This

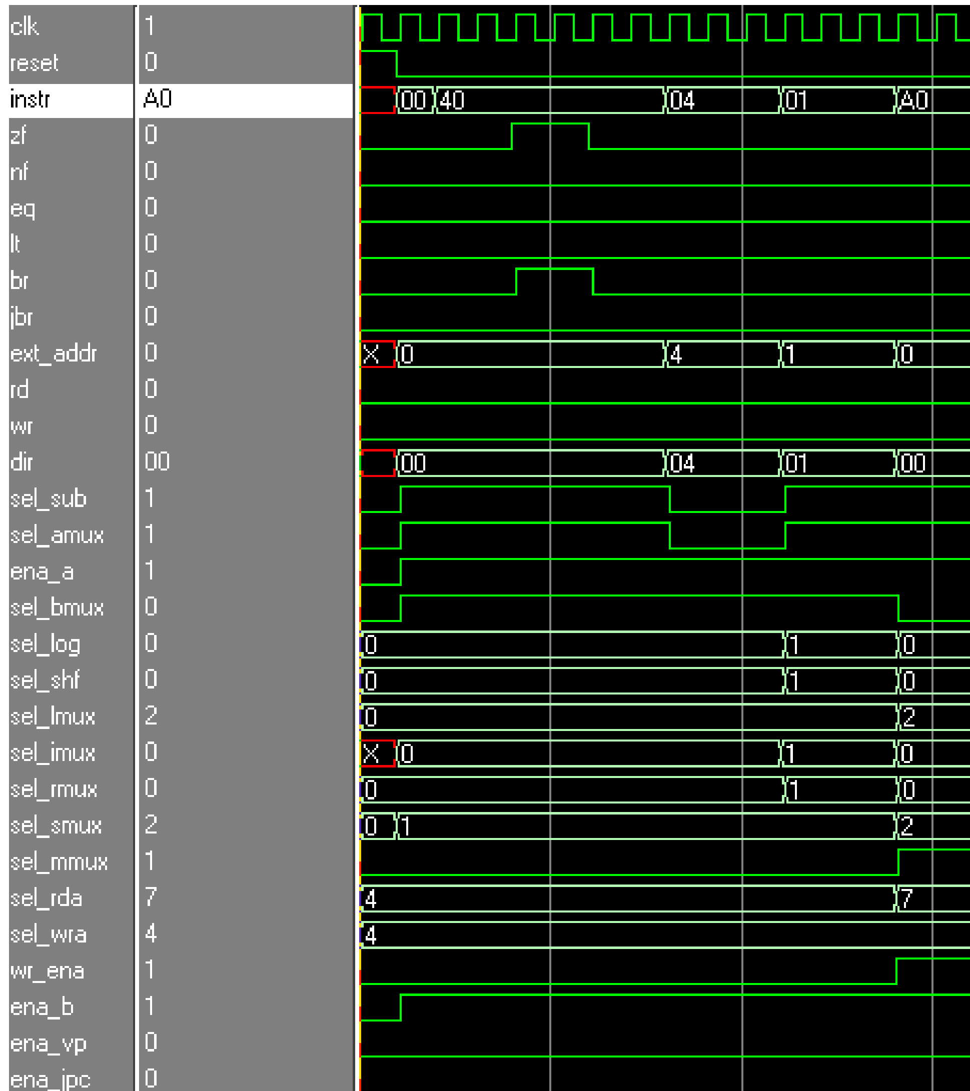

Figure 6.4: Decoder Testbench Waveform

program is a continuous loop, as shown by the read and write signals, *rd* and *write* respectively, going from one to zero and back repeatedly. Also notable is the *ext_addr* output. This shows part of the instruction code coming from the ROM, the looping is visible here too, in the form of the same outputs being shown at regular time intervals. The segment of code depicted here is shown in Appendix A. The program attempts to continuously load values from registers located on the NAND Flash memory unit into the *aout* and *bout* outputs. Due to the NAND memory being disabled, this results in both *aout* and *bout* being equal to &H4000. They are then combined with a bitwise

AND operation, shown by the one in *ext_addr*, to put an &HFFFFFFFF into *aout*, hence the toggle in the result and more values are loaded. The four in *ext_addr* represents the 'branch if zero' command. At no point, shown in this waveform, is the output *aout* equal to zero. Therefore, the branch will not be taken, and the program loops back to begin again at the command which sets *ext_addr* to zero, this is actually the nop command. The *ext_addr* only represents 3bits of the 10bit instruction. Therefore, the most significant bit is missing in the hexadecimal digit. The zero representing a nop command should actually be an eight except that it has lost its MSB. The nop command is represented by "0010000000" in binary or &H080 in hexadecimal. *Ext_addr* holds bits four to seven of this instruction. This makes it easier to tell what part of the ROM's program the JOP is currently executing and thus aided testing. All of the instruction set is acted upon in the core itself save for bits four to seven which also encode a separate instruction for the extension component located outside the core.

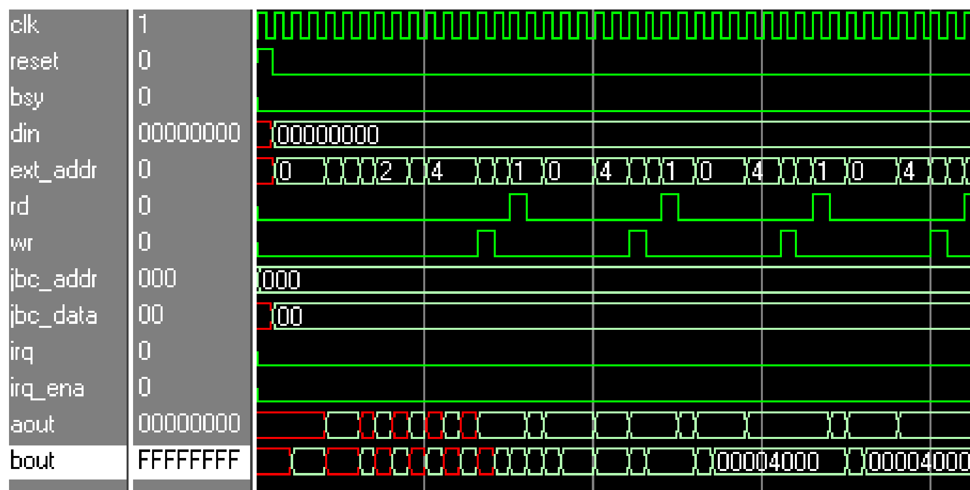

Figure 6.5: Core Testbench Waveform

6.5 Extension Testing

After the Core was complete and proven to be functioning correctly, the extra pieces of the JOP had to be tested. The Extension component was one of these. It provides an interface between other components and the Core (Section 3.3). It does this using the address mapping shown in *Table 3.3*. Extension also implements a Booth Multiplier. An example of the multiplier in motion is shown in the waveform in *Figure 6.6*. *Ain* is loaded into *dout*

and multiplied with *bin* to give the result &H3380. The multiplication takes as many clock cycles to complete as there are bits in the operands, in this case: thirty-two. The read and write inputs, as well as *ext_addr* which is really the command signal for Extension, have been set here accordingly to multiply *ain* and *bin*. All the other input and output signals are set to zero to allow this computation to take place. All other data locations not directly impacting the current computation are assigned random values.

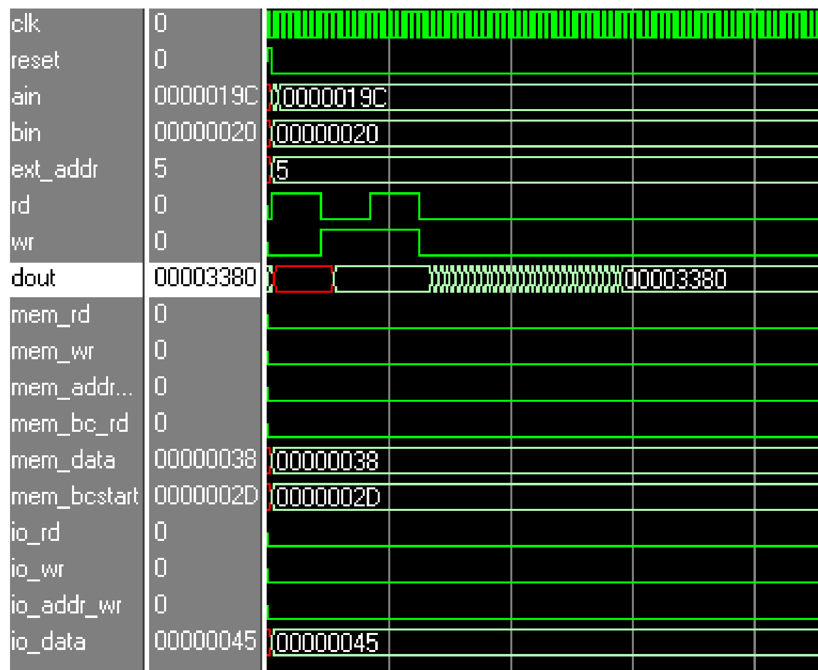

Figure 6.6: Extension Testbench Waveform

6.6 External Memory Testing

Due to a decision having been taken that the External Memory was unnecessary, it was not tested as extensively as the other components. The waveform shown in *Figure 6.7* represents the skeleton code of the memory. The RAM and ROM originally loaded into it have been removed and now output only zeroes. The *mem_bsy* signal, which is the important output the memory is still available for, is still being set when a read is occurring. This signal helps implement interrupts in the system.

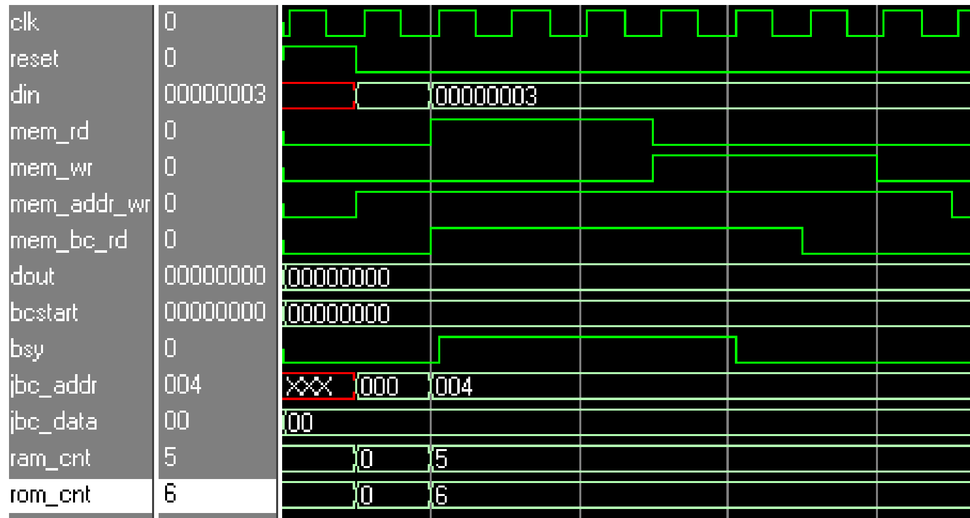

Figure 6.7: External Memory Testbench Waveform

6.7 IO Testing

The IO component is highly important as the system must be able to communicate with the outside world. *Din* provides the command for the components of the IO system; the UART and the Cnt. Due to the least significant bit of *din* being a zero, the output *dout* will be set equal to the clock counter, *clock_cnt*, in the Cnt component. This means that *dout* will increment at each clock cycle, and ensures that the IO component is doing what it should be according to the address mapping in *Table 3.5*. The signals *l*, *r*, *t* and *b* are all set to 'Z', i.e.: left unassigned. These signals are intended for direct mapping to the hardware of the Cyclone board and are unnecessary for the Virtex II ones.

6.8 Set RAM Testing

The Set RAM component, though not part of the JVM, still underwent testing once it was written to verify it was functioning as expected. As well as testing on the board, the RAM setter was also tested on the simulator. Its job was to set write signals for the on-board RAM and send values to it (Section 4.6). It's visible from the waveform in *Figure 6.9*, that on every second clock pulse, a new value is fed from the virtual RAM in this component to the output. This will feed it through to the on-board RAM.

The testing on the board was quite simple. When the program was synthesised, and made into a bitstream file (Section 5.3), it was loaded onto the

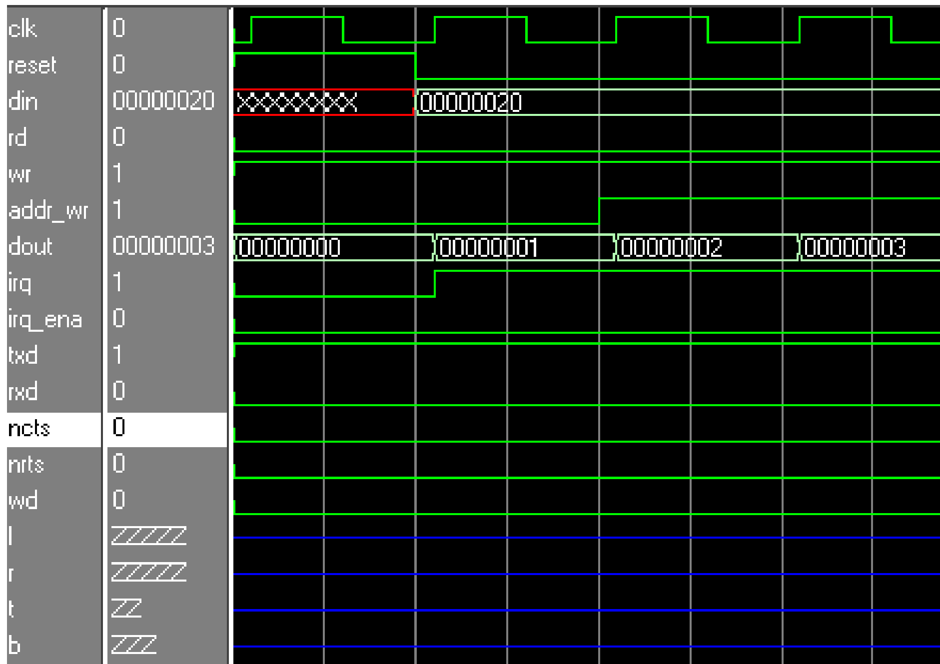

Figure 6.8: IO Testbench Waveform

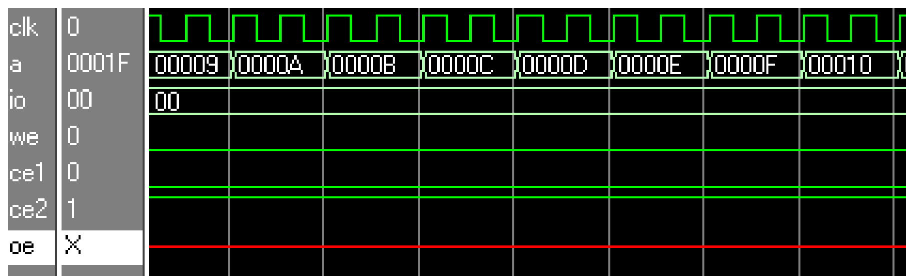

Figure 6.9: RAM Setter Testbench Waveform

FPGA. When the program finishes running correctly, i.e.: the RAM is set, the program lights the LEDs to “01010101”. By observation of the board and waveform, the RAM setter was proven to be running correctly.

6.9 8bit RAM Testing

This particular RAM was written to interact with the external RAM in the transition stage from Altera to Xilinx mentioned in Section 4. It therefore takes write enable signals, etc. directly from the JOP to be fed to the on-board RAM and routes responses accordingly. This RAM is 8bits wide but

accepts data which is 32bits wide. As discussed in Section 3.2.4, the 32bits are split up and fed through on four consecutive clock cycles. This requires two clock signals coming into the RAM, *clock* being four times slower than *clk*. The slower clock cycle is the system clock, the RAM only reads or writes one 32bit value during each of these clock cycles. During the *clk* signal, it reads or writes four 8bit values per cycle.

The first half of the waveform in *Figure 6.10* is reading, the second half writing. This RAM holds no data hence *q*, the output data going to the external RAM, remains unassigned. *A*, the address output, is loaded with the value of the read or write address multiplied by four. The read and write addresses input are intended for 32bit values, therefore if four 8bit values are to be accessed, the addresses must be multiplied by four. *A* is then incremented for the next three clock cycles so that the data is loaded into, or taken from, four consecutive memory locations. The values in *A* are changed when *we*, write enable, goes high to be four times the write address rather than four times the read one. The RAM will only begin to read or write on the next long clock cycle. Shown in the waveform is a time when the write enable signal goes high in the middle of a read. The read completes and on the next long clock cycle IO gets loaded with the 32bit data fed in, which has been split up into 8bits. In this case, the hexadecimal value &H06542435 is split up into four pieces, which are output on every clock cycle as first &H06, then &H54, next &H24 and finally &H35. This proves that the RAM interface is responding correctly to signals from the JOP and outputting the expected signals and data for the external RAM.

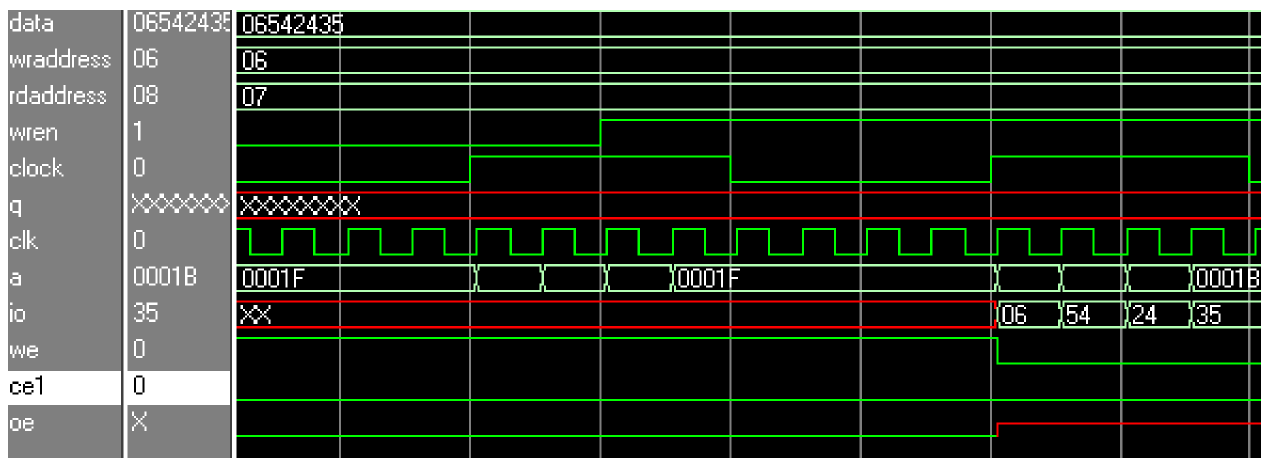

Figure 6.10: New RAM Testbench Waveform

6.10 JOP Testing

Most of the testing of the JOP took place on the board. The testbench waveform of the JOP did not show much of interest because its outputs and inputs were intended for the board. To avoid confusion, this report will forgo description of multiple versions of the JOP and skip straight to the final version. The final version of the JOP implements the shorter RAM and the JOP together, as described in Section 4.8. Testing this took place on the board. Specific testing outputs were linked to the LEDs on the board. These came directly from the Core. The resulting LED combinations were then compared with the waveform in *Figure 6.5.* When it was discovered that the completed JVM was looping through the correct program located on the virtual ROM it was evident that it was all working properly. The board clock speed was slowed down to make these results observable, and it was this version which was demonstrated. The version with an increased clock speed will be attached in disk form at the back of this report.

Chapter 7

Future Work & Conclusions

7.1 Future Work

7.1.1 Class Loader

Class loaders can get very complex, generally too complex for embedded systems. They require resolution and verification of classes before they are loaded into the execution engine. They tend to consume large amounts of memory and the upper bound of execution time is near impossible to predict[14].

However, ignoring this, having something which will dynamically load whichever classes are currently needed would be very helpful for the JVM. It would speed up operation, and provide functions and libraries previously unavailable. Though outside the current scope of the project, a class loader could be envisioned for future versions of the JOP, although care must be taken to design one which would not eat through the resources of the hardware.

7.1.2 Compiler

Most commonly used with JVMs is JIT* compilation. These compilers are basically code generators which will convert bytecode into native quickly and efficiently. They are embedded into the execution engine of the JVM. Code is compiled just as it is about to be executed, hence being a *'Just-In-Time'* compiler[1].

As mentioned in Section 2.2.1, *kissme* is currently developing a JIT compiler named *cavalry*, because the cavalry always arrives "just in time". *Cav-*

*Just-In-Time

alry simply does a fixed substitution for each piece of bytecode in a method without properly converting operations, a direct and literal translation.

The JIT is optional. It currently is not implemented and is not needed to run the bytecode on any specific Operating System architecture[16]. However, it would speed up bytecode translation to machine code immensely. These optimizations would not negate the JVM's platform independence nor any of the inherent benefits of its code[12].

7.1.3 Editor

An editor is a simple addition to the JVM. An implementation of notepad or vi could be used, the only thing really needed is somewhere to type the code. Complexities arise as the programmer requires more functionality. It could be as common as having a spell checker or even up to having a syntax checker implemented. The editor could have specific functionality for the JVM code, such as simulating how the hardware would react to the program.

The main difficulty, however, arises from the lack of an interface on the hardware. Therefore, the editor would have to be implemented on the host computer. A GUI[†], which will be covered in the next section, could help refer back to the JVM.

7.1.4 GUI

A GUI could be written for the JVM and implemented on the host computer. Both the editor and the GUI are add-on extras which could be provided by external vendors. However, having one developed specifically for this project could provide some interesting and helpful features.

7.1.5 External Memory

As remarked in Chapter 4, the External Memory unit is almost obsolete. The memory units it referenced have been removed and would need to be replaced, using Xilinx architecture, to re-enable it properly. This means that interrupts, etc. are now disabled.

It would be preferable to have these fully functioning, therefore the memory unit should be returned to its former capacity. This would have been accomplished in the course of this project except, due to time constraints, other areas had to be prioritised leaving no time to integrate new external memory units. Working with the internal RAM and planing to move it to

[†]Graphical User Interface

an external RAM on the board meant little time was spent changing the External Memory unit. It was believed then that it would not fit onto the board. Now that the RAM is once more an internal unit, this is no longer a problem.

7.1.6 Threads, Interrupts and Scheduling

This section can be linked to the previous one. Interrupts, threads and scheduling tools are already available on the JOP. However, they were disabled along with the External Memory unit. When this memory unit is re-enabled, it is likely that most of the interrupts and other features will work too, though some debugging may be required.

7.2 Conclusions

The goal of this project was to implement a Java Virtual Machine on a hardware board used by students of Trinity College Dublin. Throughout the previous months, much research, effort and time has gone into this project and there is now a JVM implented on one of the Virtex II boards. As it has been noted previously, this is just one part of an ongoing project to provide these boards with as much functionality as possible (Chapter 1).

The code used was downloaded from GPL licensed software online[14] (Chapter 2) but had to be changed significantly to run on the available hardware. This generated the greatest amount of work for the project and thus constituted its bulk. The code itself and all the changes to it are covered in Chapters 3 and 4. These chapters describe the code layout and create a visual image of JVMs in action.

Information on how to get all of this to the board is described in Chapter 5. Many different tools were combined to convert from a .vhdl file a .bit one with all of the appropriate settings to be loaded into the FPGA. This is discussed in Chapter 5 in addition to a detailed description of the board itself and all components fitted thereon.

The testing and evaluation of the hardware and software is covered in Chapter 6. This chapter builds upon all the previous ones, showing how every stage in the development process leads to another until the final stage where the code is on the board and working correctly.

This project, though at times frustrating, was overall a great learning experience. Virtual Machines and the low-level basics of Java were not covered in the course of a B.A.(Mod) Degree in Computer Science in Trinity College Dublin. This project thus afforded me with a great opportunity to build on

and independently enhance the knowledge I had in these areas. As well as this, previously unknown parts of the VHDL programming languages were uncovered. VHDL scripts written for Altera software are quite different to Xilinx counterparts. This created the feeling of entering entirely new territory when it came to debugging the code. Many new aspects of VHDL were discovered and several pieces of code had to be replaced to port between Altera and Xilinx correctly (Chapter 4).

In comparison to the rest, the Virtex II boards were easily decipherable, though this is likely due to some very timely help given by Ross Brennan. Ross explained the board's inner workings and gave tutorials on its programming. This aid proved invaluable when integrating the software with the hardware.

To sum up, the goal of this project was met and surpassed. While working on it I learned many new things which would have otherwise gone uncovered. I gained a lot of knowledge and hopefully my contribution will help future students doing hardware subjects and working with the HAL in Trinity College Dublin.

Bibliography

[1] Wikipedia. http://en.wikipedia.org/.

[2] Altera corporation. http://www.altera.com/.

[3] BRENNAN, R. Design and Evaluation of an FPGA Based Microprocessor Project Board. Final Year Project.

[4] BRENNAN, R. Documentation of Virtex II Prototyping Boards. http://-www.cs.tcd.ie/Ross.Brennan/?sub=college.

[5] CHANG, K. C. *Digital Design and Modeling with VHDL and Synthesis.* Wiley-IEEE Computer Society Press, 1997. http://www.computer.org/-cspress/CATALOG/bp07716.htm. ISBN: 0-8186-7716-3.

[6] GARY B. SHELLY, THOMAS J. CASHMAN AND JOY L. STARKS. *Java Programming : Introductory Concepts and Techniques*, 2nd ed. Course Technology, 2003. ISBN: 0789568314.

[7] LEUNER, J., AND CRAWLEY, S. *kissme: A free Java Virtual Machine*, 2003. http://kissme.sourceforge.net/.

[8] MANO, M. MORRIS. *Digital Design*, 3rd ed. Prentice Hall, 2002. ISBN: 0-13-035525-9.

[9] MEYER, J. *Java Virtual Machine.* O'Reilly, 1997. ISBN: 1565921941.

[10] MEYER-BAESE, U. *Digital Signal Processing with Field Programmable Gate Arrays*, 2nd ed. Springer, 2004. http://hometown.aol.de/-uwemeyerbaese/XilinxExamples2e.htm. ISBN: 3-540-21119-5.

[11] MLYNEK, D., AND LEBLEBICI, Y. Design of VLSI Systems, 1998. http://www.vlsi.wpi.edu/webcourse/ch06/ch06.html.

[12] NAUGHTON, P. *The Java Handbook.* Osborne McGraw-Hill, 1996. ISBN: 0078821991.

[13] SAULPAUGH, T., AND MIRHO, C. *Inside the JavaOS Operating System*. Addison-Wesley, 1999. ISBN: 0201183935.

[14] SCHOEBERL, M. JOP - Java Optimized Processor, 2005. http://www.-jopdesign.com.

[15] VENNERS, B. The Lean, Mean, Virtual Machine. *Java World* (June 1996). http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html.

[16] WINDER, R., AND ROBERTS, G. *Developing Java Software*, 2nd ed. Wiley, 2000. ISBN: 0471976555.

[17] Xilinx incorporated. http://www.xilinx.com/.

Appendix A

ROM

```
– Starts a comment
depth = 1024;
width = 10;

content

begin
0 to 1ff : 080; – nop
– //
– // JVM starts here.
– //
– // new fetch does NOT reset address of ROM =>
– // it starts with pc+1
0000 : 080; – nop // this gets never executed
0001 : 080; – nop // for sure during reset (perhaps two times executed)
–
0002 : 0c0; – ldi 127
0003 : 080; – nop // written in adr/read stage!
0004 : 01b; – stsp // something strange in stack.vhd A->B !!!
– //
0005 : 0c1; – ldi 1 // disable int's
0006 : 025; – stm moncnt // monitor counter gets zeroed in startMission
–
0007 : 0c2; – ldi 0
0008 : 022; – stm heap // word counter (ram address)
–
– //
– // start address of Java program in flash: 0x80000
```

```
– //
0009 : 0c3; – ldi 524288 // only for jvmflash useful
000a : 02c; – stm addr
–
– xram_loop:
000b : 0c4; – ldi 4 // byte counter
– ser4:
– // ************** change for load from serial line **************
000c : 0c4; – ldi io_status // wait for byte from uart
000d : 008; – stioa
000e : 0c5; – ldi ua_rdrf
000f : 0e1; – ldiod
0010 : 001; – and
0011 : 080; – nop
0012 : 040; – bz ser4
0013 : 080; – nop
0014 : 080; – nop
–
0015 : 0c6; – ldi io_uart // read byte from uart
0016 : 008; – stioa
0017 : 080; – nop
0018 : 0e1; – ldiod
–
0019 : 0f8; – dup // echo for down.c, 'handshake'
001a : 009; – stiod
–
– // ************** end change for load from serial line **********
```

Appendix B

RAM

```
depth = 256;
width = 32;
content
begin
0 to ff:12345678;
0000:00000000;
0001:00000000;
0002:00000000;
0003:00000000;
0004:00000000;
0005:00000000;
0006:00000000;
0007:00000000;
0008:00000000;
0009:00000000;
000a:00000000;
000b:00000000;
000c:00000000;
000d:00000000;
000e:00000000;
000f:00000000;
0010:00000000;
0011:00000000;
0012:00000000;
0013:00000000;
0014:00000000;
0015:00000000;
0016:00000000;
```

0017:00000000;
0018:00000000;
0019:00000000;
001a:00000000;
001b:00000000;
0020:0000007f;
0021:00000001;
0022:00000000;
0023:00080000;
0024:00000004;
0025:00000002;
0026:00000005;
0027:00000008;
0028:00004000;
0029:ffffffff;
002a:00000003;
002b:0000ffff;
002c:80000000;
002d:000000ff;
002e:0000001f;
end;