A Self-Reconfiguring Platform For Embedded Systems

by

Santiago A. Leon

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
In partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Engineering

Dr. Mark T. Jones, Chair

Dr. Peter M. Athanas

Dr. James M. Baker

August 24, 2001
Blacksburg, Virginia

Keywords: Reconfigurable Computing, FPGA, JBits, Java Virtual Machine, uClinux

Self-Reconfiguring Platform For Embedded Systems

Santiago A. Leon

Committee Chairman: Dr. Mark T. Jones
The Bradley Department of Electrical and Computer Engineering

(ABSTRACT)

The JBits Application Programming Interface has significantly shortened FPGA reconfiguration times by manipulating the configurable resources of the FPGAs directly under software control. The execution of JBits programs, however, requires a Java Virtual Machine to be implemented on the platform where the configurations will be modified. This presents a problem for embedded systems where a microprocessor to run a Java Virtual Machine may not be available or desirable. This thesis discusses the implementation of a FPGA platform that allows the execution of JBits programs, effectively changing the configuration of a FPGA within a FPGA. This thesis also presents a four step developing and testing strategy for JBits programs that are intended to run on this FPGA platform.

Dedicated to my parents and heroes, Guillermo León and Piedad Alvarez

Acknowledgements

First of all I would like to thank Dr. Mark Jones for his patience, guidance, and support throughout my work in the CCM Lab and for serving as the chair of my graduate committee. I am very thankful to Dr. Peter Athanas for the encouragement, suggestions and technical support. It has been an honor and a pleasure to work for Dr. Jones and Dr. Athanas in the CCM Lab. My thanks also go to Dr. Mac Baker for serving on my graduate committee and for taking the time to read this thesis.

I would like to thank Rüdiger Jordan, Luke Scharf, Ryan Fong, Scott Harper, Jonathan Ballagh, Zahi Nakad, and Shashank Mehrotra from the CCM Lab for all the technical help and for making the long hours in the lab much more enjoyable.

I am very grateful to Adela Dalmau for always being there for me, Katie Rask for being such a good roommate and friend, and Dennis Collins for taking the time to review this thesis.

I would also like to thank Carolina Reyes for the encouragement, inspiration, and for making me the person that I am.

Last, but not least, I thank my parents and brother for not giving me away for adoption when I started my Engineering career at age five by taking apart the electronic appliances around the house.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Field Programmable Gate Arrays (FPGAs) have been typically used in industry for prototypes and other applications where their reconfigurable nature is exploited. This reconfigurable feature of FPGAs, however, has only been used for debugging and upgrading purposes, and not as an inherent part of the designs. There have been many approaches designed to exploit this new paradigm, the simplest being a pool of configurations that can be swapped in and out, giving the appearance of having significantly more hardware than is actually present on the FPGA at any given instance in time [1]. Another approach, which can be combined with a configuration swapping design, is to modify the configurations at run-time according to the specific current needs.

To take advantage of the run-time modification approach, the traditionally lengthy synthesis process had to be shortened. This issue has been addressed by the JBits Application Programming Interface (JBits API), a set of Java classes that permit all

configurable resources of a Xilinx FPGA to be individually programmed under software control [2]. The inherent advantages of Java, such as high-level constructs, platform independence and GUI integration have made JBits a very flexible and useful tool. The execution of Java programs, however, requires a Java Virtual Machine to be implemented in the platform where the configurations will be modified. This presents a problem for embedded systems where a microprocessor to run a Java Virtual Machine may not be available or desirable.

Such is the case for the Secure Hardware Project at Virginia Tech's Configurable Computing Laboratory [3]. In this type of system, it is desirable to hide all of the implementation of the design inside FPGAs to make it difficult to illicitly acquire hardware design information from the system [3]. To perform reconfigurations in this type of embedded platform, it is necessary to execute a JBits program inside a FPGA. By performing reconfigurations inside the FPGA, not only the hardware implementation and software for reconfiguration is secure, but the system can also be loaded into the design only when it is necessary to perform a reconfiguration, saving valuable resources.

1.2 Contributions of this Work

The main contribution of this work is the design and development of a FPGA platform that will execute JBits programs, effectively changing the configuration of a FPGA within a FPGA. This work is considered a platform because it was necessary to develop and modify ports of a microprocessor, an operating system, a Java Virtual Machine, as well as some standard Java API classes. Another aspect of this contribution

is the presentation of a four step developing and testing strategy for JBits programs that are intended to run on this FPGA platform.

1.3 Thesis Organization

Chapter 2 of this thesis provides the background material on which this work is based. Chapter 3 describes the port of the LEON processor to a FGPA board. Chapter 4 details the contributions to the LEON processor port of the uClinux operating system. Chapter 5 describes two ports of a Java Virtual Machine: one to Linux running on a standard PC, and one to uClinux running on the FPGA board, as well as some standard Java API class developed to support JBits. Chapter 6 presents some applications and programs running on the platform that followed the development strategy. Finally, Chapter 7 concludes with an overview and suggestions for future work.

Chapter 2

Background

This work was conceived using a top-down approach to the problem. To run a JBits program, a Java Virtual Machine is necessary. A Java Virtual Machine requires an operating system for low-level functions. The operating system for running the Java Virtual Machine runs on a microprocessor that is synthesized as part of the FPGA design. Finally, the FPGA is the processing element of a board that contains other components such as memory and I/O connectors. This chapter, however, presents the topics that form the basis for this work in a bottom-up manner. First it describes the FPGA board used, then the microprocessor, followed by the operating system, Java and the Java Virtual Machine, and an overview of the JBits API. The last section of this chapter describes previous work related to this thesis.

2.1 SLAAC1-V board

The SLAAC1-V board was developed by the University of Southern California, Information Sciences Institute as part of the SLAAC project, whose objective is to develop and deploy a system-level, open, distributed, heterogeneous adaptive computing architecture standard [4]. Figure 1 shows the overall architecture from the user's perspective. The board is a 64-bit PCI card with the following features [4], [5]:

- Three user-programmable Xilinx Virtex XCV1000-6 FPGA's (X0, X1, and X2)
- One user-programmable Xilinx Virtex XCV200-6 FPGA (Configuration Controller)
- 10 MB of SRAM organized as 2 MB for X0, 4 MB for X1, and 4 MB for X2. Additionally, X0 can access all of the memory by performing bank switching.
- One 72-bit wide ring bus around X0, X1, and X2.
- One 72-bit wide crossbar bus connected to X0, X1, X2, and three QC-64 external connectors.
- 2-bit wide handshake lines connecting X0, X1, and X2 to each other.
- 3-bit wide handshake lines between the PCI Interface on X0 to both X1 and X2.
- Two 68-bit wide 1-deep (mailbox) FIFOs and two 68-bit wide 256-deep FIFOs.
- PCI bus directly connected to X0.

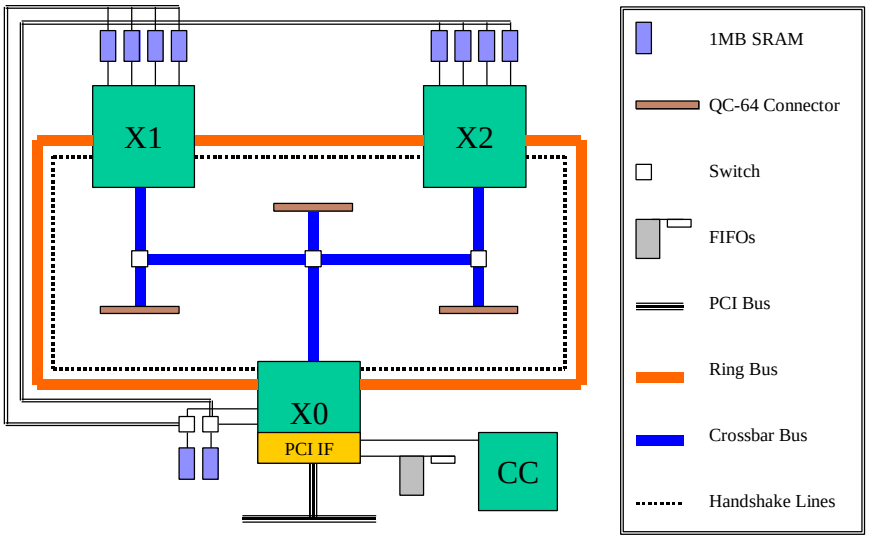

Figure 1 SLAAC1-V block diagram

Figure 1 shows how X0 performs a special role in the SLAAC1-V board. It contains the PCI interface, and it is used to control the data flow from the host PC to the board. This data flow is mainly accomplished using the hardware queues (commonly referred to as FIFO's), memory banks, and registers in the PCI interface.

All of the functions provided by the host program API are performed through registers in the PCI interface, including access from the host to memory banks, access to the FIFOs, and access to the FPGA configurations. The FPGAs on the board are programmed through the Configuration Controller that is directly connected to X0's PCI interface. It is also possible to set the clock speed, start and stop the FPGA clock, single step, and toggle the FPGA reset signals from the host program.

To implement a user design into a FPGA, the top level VHDL entity needs to be instantiated in one of a set of predefined VHDL files. These files describe the board, memory interface and provide a special frame for the user design [6]. This is especially important for X0 because the PCI core needs to be instantiated within the user design. The manufacturer of the board also provides the constraints file needed for pin assignments and timing constraints.

2.2 LEON processor

LEON is a synthesizable VHDL model of a 32-bit SPARC compatible processor developed by the European Space Agency (ESA) [7]. The source code is freely

distributed under the GNU LGPL license. Compilers, operating systems, and simulators are available from ESA and third party vendors.

2.2.1 VHDL model

The VHDL model of the LEON processor supports all of the main features of the SPARC V8 architecture. Even though it has been extensively tested against the SPARC V8 architecture manual and the IEEE-P1754 (SPARC) standard, it has not been certified by SPARC International as being SPARC V8 compliant [7]. The processor is extensively configurable and can be implemented on both FPGAs and ASIC technologies. The only technology-specific mega-cells needed are ram cells for caches and the register file [7]. The processor has an integer unit with a 5-stage pipeline, separate direct-mapped instruction and data caches, hardware multiply, divide, and MAC units. The distributed source code also includes [7]

- a programmable 8/16/32-bits memory controller for external prom and static ram,
- a controller for AMBA-2.0 AHB and APB buses,
- an interrupt controller,
- two timers,
- two programmable UARTs,
- a watchdog unit,
- a 16-bit I/O port,
- an interface for FPU unit or coprocessor, and
- a bootstrap loader

These on-chip peripherals are connected to the AMBA bus shown in Figure 2.

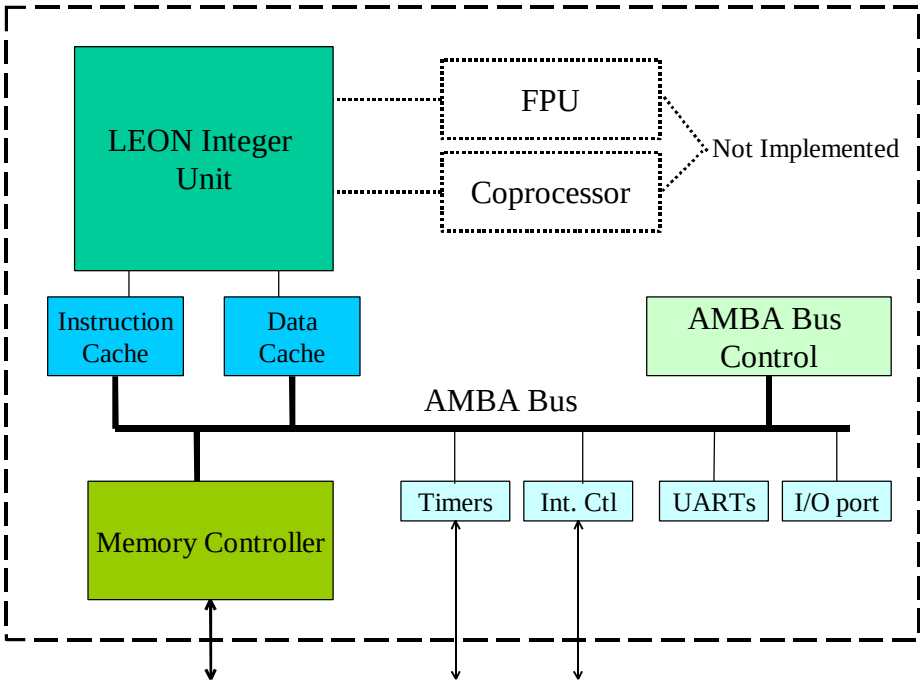

Figure 2 LEON block diagram

The memory controller and its interface to external memory devices are of special interest in this work. The memory controller supports three types of devices: PROM, SRAM, and I/O, which are mapped to different address areas as shown in Table 1. The data buses can be configured to support 8, 16, and 32 bit devices.

Device	Address range	Size
PROM	0x00000000 - 0x1FFFFFFF	512MB
I/O	0x20000000 - 0x3FFFFFFF	512MB
RAM	0x40000000 - 0x7FFFFFFF	1GB

Table 1 LEON Memory Address map

As Figure 3 shows, the memory controller has internal support for up to four banks in the SRAM area each one having an independent Chip Select (RAMSN), Output Enable (RAMOEN), and Read/Write (RAMWEN) signals. The PROM area has support for up to two banks with independent Chip Select (ROMSN) signals. The memory mapped I/O space is controlled by a single Chip Select (IOSN) signal. The PROM and I/O areas share Output Enable (OEN) and Write Enable (WEN) signals. Memory bank timing, size, and data widths are configured through two memory-mapped registers in the processor.

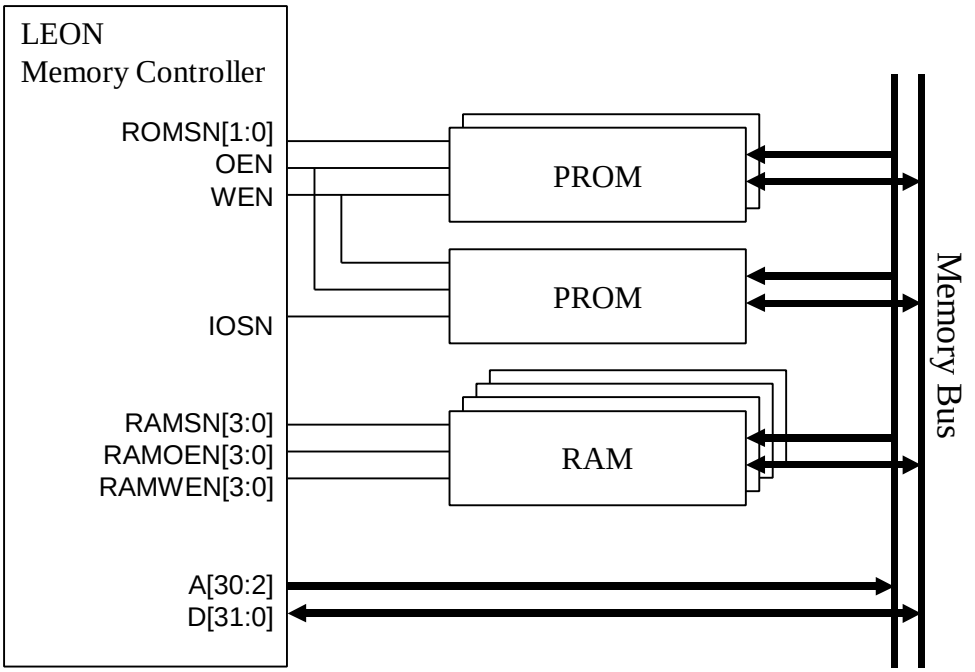

Figure 3 LEON Memory Interface

2.2.2 Compilers

ESA provides a modified version of the GNU C/C++ and an ADA cross-compiler for the LEON SPARC core. The compiler runs under Linux, Solaris or Windows and includes the necessary programs to develop stand-alone programs such as a linker and

assembler. The package also includes a POSIX compliant C-library, with standard I/O to a UART, as well as a Math library. The LEON/ERC32 Cross Compiler System (LECCS) package includes

- a custom GNU C/C++ compiler,
- binary utilities such as linker, assembler, and archiver,
- a standalone C-library,
- a real-time kernel (Real-Time Executive for Multiprocessor Systems or RTEMS),
- a boot-prom utility,
- a GNU debugger with Tk front-end,
- a graphical user interface for the GNU Debugger (Data Display Debugger or DDD), and
- a remote target monitor.

Because the LEON processor is SPARC V8 compatible, any SPARC compiler can be used to generate programs. The standard distribution of the GNU Compiler Collection (GCC) and GNU binary utilities (BINUTILS) include support for SPARC processors. They are traditionally used to compile programs for Linux running on a Sun workstation, but they can generate programs for the LEON processor. In addition, the standard distribution of GCC can generate any binary file format, as opposed to the LECSS version that only supports “a. out” and “bin” binary formats.

GCC includes compilers and pre-processors for C, C++, Fortran, Objective C, and other languages [8]. BINUTILS includes an archiver, a linker, an assembler, an assembly

preprocessor, and utilities to transform between different binary file formats [9]. Note that the SPARC build of these utilities has the prefix “sparc-linux-“ to denote that the utility is a cross compiler for a SPARC processor running the Linux operating system.

2.2.3 Operating Systems

The development of applications more advanced than stand-alone C programs requires the support of an operating system [6]. For this work, some of the most relevant advantages of having an operating system are integrated trap handling, filesystem support, device driver support, and multitasking.

The LECSS package includes the RTEMS real-time kernel as an API implemented into libraries that are linked with the user program. The LEON port of RTEMS supports multitasking, preemptive scheduling, POSIX threads, interrupt handling, dynamic memory allocation, and an In-Memory Filesystem. Inter-process communication can be implemented through semaphores, condition variables, message queues, events, and signals [10]. Because of the impressive set of features, RTEMS has been extensively used in defense systems with ports to several processors available.

An alternative operating system for the LEON processor is uClinux. The port is not yet complete and is discussed in later this chapter and in Chapter 4.

2.2.4 Simulator

ESA has made available to the public an evaluation version of TSIM, a SPARC architecture simulator capable of emulating LEON-based computer systems [11]. This evaluation version has the following features:

- it emulates the full functionality of the LEON VHDL model, including cache memories, UARTs, timers, and the memory controller [11],
- the UARTs can be mapped to a Linux device. Usually the first UART is used as a console, so by default it is mapped to the standard I/O,
- simple debugging functions are available, such as single-stepping, breakpoints, memory and registers inspection, and a disassembler, and
- it can be connected to the GNU debugger (gdb) acting as a remote target and supports all of gdb's debugging requests.

Unfortunately, the evaluation version has some limitations that the commercial versions do not:

- it supports only 2^{32} ticks of simulation time as opposed to 64-bit in the Standard and Professional versions,
- the amount of simulated ROM is fixed to 2MB and 4MB for RAM (in the commercial versions, the amount of simulated memory is a command line argument and is only limited by the resources of the host computer),
- the commercial versions support loadable modules for emulation of user-defined I/O devices,

- the sizes of the caches are fixed to 4 KB with 4 words per line, in contrast to a fully configurable size and depth, and
- it is not possible to display the cache contents in the evaluation version.

The limitation of 2^{32} ticks of simulation time translates to approximately five minutes with the processor running at 14MHz. This limitation became an issue when executing large JBits programs. However, there is an older version of TSIM called SPARC Instruction Simulator (SIS). This program does not have all the debugging features, namely the connectivity to *gdb*, but it does support 64-bit simulation time.

2.3 Linux / uClinux kernel

It is common knowledge that Linux is a very popular UNIX-like operating system distributed under the GNU General Public License. The Linux operating system is composed of several components, all relying on a single basic element, the kernel. The Linux kernel is a monolithic kernel, in the sense that it is a single very large and complex program. It carries out a large number of tasks including managing processes, filesystems, memory, hardware, and devices.

uClinux is a derivative of the Linux 2.0 kernel intended for processors without Memory Management Units (MMUs) [12]. The uClinux operating system's main targets are microcontrollers (μCs) for embedded systems. The first successful target was the 3Com PalmPilot; since then, there have been many successful ports to a wide variety of processors such as the ARM 7TDMI, the Intel i960, the ETRAX, Motorola's DragonBall,

the Coldfire, the QUICC, and many other 6800 and 68000 series processors. The LEON port of uClinux is discussed in Chapter 4.

The source code for the kernel has been rewritten to reduce the memory requirements making the uClinux kernel much smaller than the original Linux 2.0 kernel. The standard C library had to be modified as well to construct more compact user programs. Regardless, uClinux retains the main advantages of the Linux operating system: stability, network capabilities, and filesystem support [12].

Multitasking is supported in uClinux, but there are some limitations that are imposed by not having a MMU [12]:

- uClinux implements *vfork()* and not *fork()*, which means that the child and the parent do not get independent copies of the variables and that the parent blocks until the child calls *exec()* or *exit()*.
- uClinux does not have autogrow stack for user programs and no *brk()*. To allocate memory *mmap()* needs to be called.
- There is no memory protection; any program can access all of the available memory, including the kernel and I/O space.

2.4 Java

Java is an architecture that is formed out of "four distinct but interrelated technologies, each of which is defined by a separate specification from Sun Microsystems:

- the Java Programming Language
- the Java class file format
- the Java Application Programming Interface
- the Java Virtual Machine" [13]

All of these technologies are relevant for this work. Sections 2.4.1 and 2.4.2 briefly describe some of the characteristics of each technology. Section 2.4.3 describes Waba, the Java Virtual Machine ported for this work.

2.4.1 Programming language and API

Sun Microsystems describes Java as a "simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, and dynamic language." [14] The language closely resembles the C++ language with multiple alterations. Some of these alterations are intended to make the syntax simpler; for example, neither templates nor enumeration types are supported. Some other alterations are intended to make the programs more robust by avoiding common programming pitfalls. Nevertheless, the greatest advantage of Java over C or C++ is portability.

There are no implementation-dependent aspects in the language specification. The language is interpreted, so there are no native dependencies of the binary format. The language specification goes so far as to specify the size of each primitive data type, as well as its arithmetic behavior [14]. In theory, once a program has been written and

compiled for a specific platform, the compiled binary can run on any platform to which the Java API and the Virtual Machine have been ported.

The Java API is a set of runtime libraries that gives the programmer a standard way to access the system resources of the host computer. The API also provides the programmer with an extensive set of commonly used data structures, algorithms, and features that make up for the simpler syntax. The programmer can assume that the class files of the Java API will be available from any Java Virtual Machine that may ever execute the program. The class files of the Java API are inherently specific to the host platform; this makes platform independence possible for Java programs. Additionally, the Java API contributes to Java's security model because there are methods of the Java API that check for permissions before they perform any potentially harmful action.

2.4.2 Java class file format and Java Virtual Machine

The Java class file format is an intermediate representation of a Java object or interface. It consists of a stream of 8-bit bytes with a simple organization optimized for transmission over a network. The first four bytes contain a magic number used to verify that the stream is a class file and not any other stream. The next four bytes contain the version of the compiler that generated the class. The next two-byte field contains the size of the constant pool, followed by a variable length field of the constant pool. The next two-byte fields contain the class flags, the index of the class name to the constant pool and the index of the Superclass name to the constant pool. The remainder of the file has the same structure for Interfaces, Fields, Methods, and Attributes: a two-byte field

containing the number of items followed by an array of the respective items [15][16]. All of these fields are parsed and executed by the Java Virtual Machine.

The Java Virtual Machine is an abstract computer system. Its primary job is to load class files and execute the bytecodes that they contain. The specification defines the following [16]:

- Data types are either primitive data types or reference types.
- Primitive data types have specific sizes and operations.
- Three kinds of reference types: class types, array types, and interface types.
- Independent Program Counter registers for each thread.
- Independent Java Virtual Machine stacks for every thread. These are analogous to a C program stack and the frames can only be pushed or popped.
- Each frame has its own array of local variables, its own operand stack, and a reference to the runtime constant pool.
- All operations between data types are done in each frame's operand stack.
- A heap that is shared among all the threads and is used as a runtime data area for all class instances and arrays.
- A method area shared among all the threads. It is used analogously to the ". text" segment in a UNIX process.
- A run-time constant pool built from the constant pools of every loaded class.
- 202 op-codes divided into arithmetic, load and store, type conversion, object creation and manipulation, operand stack management, transfer control, method

invocation and return instructions plus a breakpoint instruction, and two implementation dependent instructions.

- Specific synchronization rules about the access of variables and locks from different threads.

The specification defines many features but also leaves many choices to the designers of each implementation. For example, although all Java Virtual Machines must be able to execute Java bytecodes, they can use many available techniques to execute them [13].

2.4.3 Waba

Waba is a small, efficient, and reliable Java Virtual Machine aimed at portable devices. Rick Wild of Wabasoft developed the first version, but now it is an open source project distributed under the GNU General Public License and maintained in Sourceforge [17]. All of the source code is written in standard ANSI C, simplifying ports to new platforms. At the time this thesis was written, there were ports for PalmOS, PocketPC, and x86 running Windows. The Virtual Machine takes less than 40KB of executable code on Motorola 68K processors and about double that on x86 processors. The Virtual Machine also includes a deterministic garbage collector for heap management [17].

Part of the Waba project is the development of a set of foundation classes to access the system resources of the host computer and aid the programmer with graphics primitives. Currently, the foundation classes are [17]:

- fx: simple effects -- *Color*, *Font*, *FontMetrics*, *Graphics*, *ISurface*, *Image*, *Rect*, *Sound*, *SoundClip*.
- io: access to IO resources -- *Catalog*, *File*, *SerialPort*, *Socket*, *Stream*.
- lang: subset of the standard *java.lang* package -- *Object*, *String*, *StringBuffer*.
- sys: miscellaneous system utilities -- *Convert*, *Time*.
- ui: simple widget-oriented GUI -- *Check*, *Container*, *Control*, *ControlEvent*, *Edit*, *Event*, *IKeys*, *KeyEvent*, *Label*, *MainWindow*, *PenEvent*, *Radio*, *Tab*, *TabBar*, *Timer*.
- util: miscellaneous utilities – *Vector*.

Many classes, though slightly different and simpler, resemble the standard Java API. Some other classes were designed to take advantage of specifics of a platform and do not really have an equivalent in the Java API. For example, the *io.Catalog* class was designed to support PalmOS's catalogs and is not useful on any other platform.

2.5 JBits

JBits is a set of Java classes that provide an Application Program Interface (API) into the Xilinx FPGA bitstream [2]. The interface can use a bitstream read back from the actual hardware or generated from the Xilinx design tools. Once the bitstream is loaded, all of the configurable resources of the FPGA can be read and modified. These configurable resources include configurable-logic blocks (CLBs), routing switches, input/output blocks (IOBs), block RAM (BRAM), and routing MUXes [18].

Within JBits, classes and class constants represent the configurable resources. Some other class constants represent the possible configurations of certain resources and will become the contents of the configuration SRAM inside the FPGA. Every resource that can be configured using JBits is identified by a series of indices that reflect its position relative to the lower-left corner of the FPGA CLB array [19]. For example, the following code modifies the "F" LUT of the left slice of an arbitrary CLB (in this case the second column of the last row) to the value 0xBEEF.

```
/* set the location of the CLB */
int y = 0;
int x = 1;

/* set which slice in that CLB */
int slice = 0;

/* set the value to be written (binary of 0xBEEF)*/
int value[] = {1,0,1,1,1,1,1,0,1,1,1,0,1,1,1,1};

/* call the method */
jbits.set(y, x, LUT.F[slice], value);
```

Using the JBits API, Xilinx FPGA bitstreams can be created and modified in the order of seconds as opposed to hours when using the traditional FPGA synthesis and compilation tools. Furthermore, the time to execute a JBits program can be significantly shortened with the use of partial bitstreams [19].

Although the JBits calls perform actions at a very low level, the object-oriented nature of Java and its GUI support have been exploited to produce a small library of parameterizable, macro circuits or Cores, as well as higher-level tools such as Boardscope [19]. Boardscope is a graphical FPGA debugging environment that can be

connected through a special interface to an actual FPGA, or can run in simulation mode by loading the bitstream from a file. The tool can display the current state of the FPGA in various modes: State, Core, Power, and Routing Density. For this work, the State mode was very useful because it was possible to visually inspect the values of any LUT, flip-flop or MUX in any CLB [20].

2.6 Related Work

This section provides an overview of previous work related to aspects of this thesis. Namely, Section 2.6.1 reviews some freely available and commercial processors that can be implemented in a FPGA. Section 2.6.2 surveys different approaches to running Linux on embedded devices. Section 2.6.3 analyses other alternatives for running a Java Virtual Machine on a FPGA. Section 2.6.4 reviews previous work on self-reconfiguring platforms.

2.6.1 Soft-core processors

There have been numerous attempts to develop a processor core that can be implemented in a FPGA. This section describes some of the processor cores that are powerful enough to be part of a self-reconfiguring platform.

As part of an extensive library of cores, Altera developed Nios, a processor specifically designed for programmable logic and system-on-a-programmable chip integration [21]. The Nios processor is a pipelined, single-issue RISC processor in which most instructions run in a single clock cycle. There are two versions available, one with a

native 16-bit word size and one with a native 32-bit word size. The 16-bit version is designed to replace complex state machines as a very simple controller while the 32-bit version is designed to be a powerful computing engine [21]. There is a development kit available which includes a C/C++ compiler, a debugger, an assembler, as well as other development utilities. At the time this thesis was written, there was only a proprietary operating system ported to the Nios processor. However, there is currently an ongoing effort to port Linux to this processor.

Xtensa is a 32-bit processor architecture developed by Tensilica. The company provides software in which various parameters of a processor, such as instruction set, cache characteristics, and optional components, can be defined in a set of configuration files [22]. These configuration files can later be uploaded to Tensilica, and after one hour, a complete software development environment can be downloaded. The software development environment includes a customized compiler, assembler, debugger, simulator and real-time operating system. Once the design and features have been verified and the program executed in the simulator, Tensilica will provide full source of a Hardware Description Language. This source code can be synthesized to a FPGA or to an ASIC and the same customized tools can be used to develop the applications for the final implementation.

The base processor of the Xtensa architecture is a pipelined 32-bit RISC processor that can achieve more than 220 MIPS while running at 200MHz [22]. For 0.18-micron CMOS ASIC process, the processor consumes less than 0.4 mW/MHz and will use

approximately 0.7 mm^2 of area in the silicon wafer. There are many optional components that can be included with the base processor including a 16-bit hardware multiplier, a 16-bit DSP unit, and an on-chip debug module.

In response to the development of soft-core processors, Xilinx introduced the MicroBlaze processor. The MicroBlaze processor is a 32-bit RISC processor that supports both 16-bit and 32-bit busses and supports Block Ram and/or external memory [23]. All peripherals including the memory controller, the UART, and the interrupt controller run off a standard OPB bus. Additional processor performance can be achieved by exploiting Virtex-II architecture features such as the embedded multiplier and ALU [23]. Xilinx also provides Gnu-based tools, including a C-compiler, a debugger, and an assembler, as well as all of the standard libraries [23]. At the time this thesis was written, the MicroBlaze processor was in the beta stage of development and there was no information on the operating system.

One of the primary disadvantages of these processors compared to the LEON processor is that they are commercial products that require a fee for use. Additionally, these architectures are specifically designed for particular platforms, which limits their flexibility.

A processor core that doesn't have these disadvantages is the OpenRISC 1000 processor [24]. The fully synthesizeable code of the OpenRISC 1000 processor is freely available and was designed with an emphasis on scalability and platform independence.

The architecture consists of a 32-bit RISC Integer Unit with a configurable number of general-purpose registers, configurable cache and TLB sizes, dynamic power management support, and space for user provided instructions [24]. The core also includes a memory management unit with powerful virtual memory support, as well as peripherals for SMP support and cache coherency support [24]. A complete GNU-based development environment is available and includes a C-compiler, assembler, linker, debugger, and simulator. There is an ongoing effort to port Linux to this processor. Unfortunately, at the time this thesis was written, the OpenRISC processor was still in the testing phase and the development had slowed down significantly.

2.6.2 Embedded Linux devices

In the last few years different companies and groups have developed literally hundreds of embedded systems that run Linux or some derivative of Linux. Because the source code of Linux and its derivatives is free, there are ports of Linux for almost every type of processor for which there is a GNU-based compiler. Depending on the processor features and the application of the embedded system, either Linux, uClinux, or RTLinux is used. The regular distribution of Linux is used on systems with an MMU and for which a kernel with real-time capabilities is not required. As stated earlier in this chapter, uClinux is used in embedded systems whose processor does not have a MMU [12]. Finally, RTLinux is used in microprocessors with a MMU and where the application of the system has time-critical tasks that require a real-time kernel [25]. Additionally the RTLinux patches have been applied to uClinux to support processors with no MMU and applications that require a real-time kernel.

Because of the increasing number of new Linux embedded devices, any list of such devices will be obsolete in a short amount of time. Nevertheless, [26] has a comprehensive list of devices that gets constantly updated as well as links to the different porting projects.

2.6.3 Java Virtual Machines on FPGAs

This section describes various Java processors that can be compiled into a FPGA. Note the use of the term "Java processor" and not Java Virtual Machine because these processors are only the Execution Engine subsystem of a JVM, and do not perform the functions of the Class Loader and Security subsystems [13] [16]. The latter subsystems have to be implemented to have a fully functional JVM. Currently, there are no cores that include the three JVM subsystems, with the exception of designs that contain a Java processor and a general-purpose processor to perform the functions of the Class Loader and Security subsystems.

In 1998, Sun Microsystems introduced the first soft core of a Java processor and named it the picoJava-I [27]. An upgrade of the picoJava-I core, the picoJava-II core has a six-stage RISC pipeline with forward instruction folding and includes various improvements in performance, silicon area, and power consumption [28]. It features thread synchronization and a variety of garbage collection methods. Additionally, the picoJava-II processor supports method invocation and load hiding from local variables, streamlining object oriented programming [28]. Most of the Java bytecodes are executed

directly in hardware with the exception of the more complicated op-codes that are emulated in software. The core features instruction and data cache units, an Integer Unit, a Stack Manager, and a Bus Interface Unit. It lacks, however, memory and interrupt controllers, making it necessary to include them in either hardware or in a program of a general-purpose processor. For a licensing fee, Sun Microsystems provides a Verilog RTL model, a logic library, and the internal specifications as well as technology training and support [28].

Kim Austin of Lucent Technologies and Dr. Morris Chang from the Illinois Institute of Technology developed a fully synthesizable VHDL model of a Java processor [29]. The internal structure of the processor closely resembles the abstract components of the JVM specification. The core was intended to be a proof-of-concept, so only a subset of the Java op-codes were implemented in the processor and it does not include any kind of cache [29]. The design was verified in an Altera EPF10k20 FPGA and the core consumed less than 50% of the resources of the FPGA. The authors later developed an improvement of the design that includes a smart class loader and instruction folding [30].

In 1999 Derivation Systems Inc. (DSI) introduced the LavaCORE FPGA IP core based on its proprietary Formal Synthesis technology [31]. It is provided as a fixed core or synthesizable soft-core with a suite of tools for parameterized core generation, hardware/software co-design, co-verification, and custom Java application development [31]. The processor core is a 32-bit microprocessor and the soft-core version is targeted to the Xilinx XC4000EX FPGA architecture. There is an evaluation/development kit

available from Xilinx's IP cores library. It now supports the Virtex and Virtex-II series FPGAs [32].

Another Java processor commercially available is the Moon processor from Vulcan ASIC Inc. [33]. As opposed to the other processors reviewed earlier, the Moon processor does not require a class loader because the company provides a proprietary linker. This linker converts the object references to physical memory addresses, checks the constant pool structure, resolves the constant pool types to physical addresses, and links the program with an implementation of the Java API. Avoiding the class loader has its advantages; however, it wouldn't be possible to add any new classes to be executed once the intermediate file is generated. This can become a disadvantage when attempting to upgrade or add classes dynamically from a network.

2.6.4 Self Reconfiguring platforms

This section provides an overview of some projects that exploit the flexibility of reconfigurable devices to build a self-reconfiguring platform. Note, however, that the platforms only perform reconfigurations on specific previously defined parameters and not on the entire platform.

Good examples of that type of platforms where specific parameters are self-reconfigured are systems that use neural networks. For many of the applications of neural networks the performance of software simulation running on a sequential computer is not sufficient to be practical [34]. To solve this problem, James G. Eldredge

and Brad Hutchings of Brigham Young University developed a hardware implementation of the neural backpropagation algorithm [34]. This implementation reconfigures the same FPGAs for multiple phases of the algorithm giving the impression of having significantly more hardware available. The reconfigurations are done at runtime and depend on the data and data flow generated by the algorithm, effectively self-reconfiguring the platform.

Tadayoshi Horita and Itsuo Takanami developed a platform in which the routing of a mesh-array can be reconfigured to alternate routes when there is a fault in one of the links [35]. Masaru Fukushi also used FPGAs that self reconfigure to solve the same problem with excellent results [36]. Both of these platforms used neural networks algorithms and modified different parts of the design at runtime.

Another interesting approach is the Dynamic Instruction Set Computer (DISC), developed by Michael Wirthin and Brad Hutchings of Brigham Young University [37]. Using FPGA partial reconfiguration, this computer pages in and out instruction modules as demanded by the executing program. The global controller unit of the processor executes simple arithmetic operations and cycles the custom instruction modules. Once the custom module is swapped in, it performs the operation when a specific op-code found in the program is intercepted by the module's decode unit. The controller can have a large number of modules stored in memory and cycles them as required by the program, effectively self-reconfiguring the platform at runtime.

Chapter 3

LEON processor port to the SLAAC1-V board

This chapter explains the various steps that were performed to implement the LEON processor on the SLAAC1-V board. As explained in Chapter 2, the VHDL model of the LEON processor includes many on-chip peripherals allowing for an easy implementation of a complete solution on any board with only a FPGA, RAM, and serial I/O. The initial tests of the processor were made on a small stand-alone XESS XSV300 board that allowed probing of external signals for easy debugging. Once the tests on this board were satisfactory, the port to the more complicated SLAAC1-V board started, focusing primarily on memory access (Section 3.1) and serial I/O (Section 3.2). The work explained in this chapter was developed in cooperation with Rüdiger Jordan of the Darmstadt University of Technology [6].

3.1 Memory access

The original implementation of the port developed for this work had the LEON processor synthesized into X0 of the SLAAC1-V board. This decision was made because X0 can access all of the memory in the board by performing bank switching, as explained

in Chapter 2. It was a simple task to add a decoder that will drive the bank switching signals of X0. However, the bank switching takes an additional cycle that is not documented in the User's Manual [5]. The additional clock cycle makes it impossible for the processor to work with the 10MB of memory because the memory controller expects a fixed number of wait states [7]. Nevertheless, X0 can access its own 2MB of RAM with a fixed number of wait states, so they were assigned to the first two banks of the LEON's memory controller.

A problem that was encountered is that even though each RAM bank on the SLAAC1-V board has a bi-directional hardware data bus, the SLAAC1-V VHDL interface provides separate input and output data buses [5] [6]. This feature is necessary to support the bank switching of its X0's RAM banks with X1's and X2's RAM banks. An easy way to solve this problem would be to add additional multiplexers that would select the data based on the Read/Write bit. However, this approach will increase the time delay of the data, forcing the processor to run at a lower clock rate. Therefore, it was decided to modify the LEON processor's memory controller so that it provides different input and output data buses to match the SLAAC1-V interface. The task did not require significant modifications to the source code because the memory controller has separate buses internally which are merged by tri-state drivers in the I/O pad interface.

Some additional wrapper code was needed because the LEON processor should be able to access all SRAM banks and additional memory mapped I/O devices through the same address and data buses. The wrapper code primarily consists of the FIFO serial

interface (explained in Section 3.2) and multiplexers controlled by Chip Select signals. All of the other control lines are output signals that were connected directly to the corresponding RAM bank lines. A block diagram of the wrapper is shown in Figure 4.

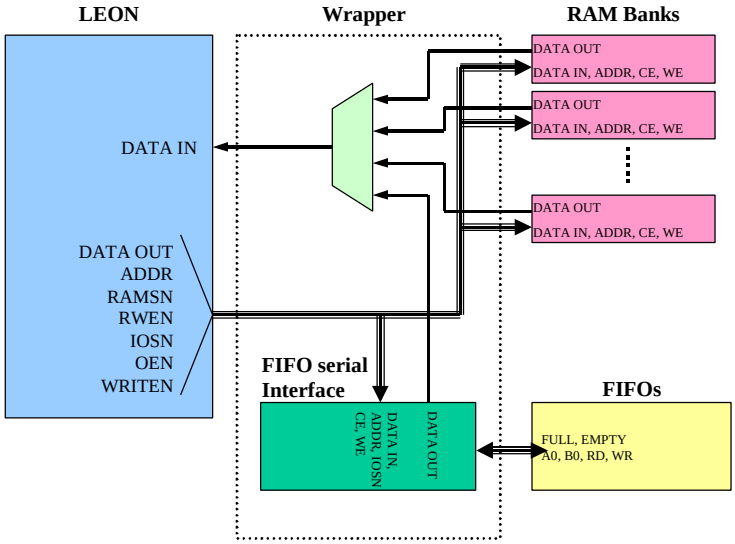

Figure 4 Block diagram of the LEON – SLAAC1-V wrapper

The implementation of the LEON on X0 used only two RAM banks of the memory controller, one for each 1MB RAM bank attached to X0. Nevertheless, the wrapper code can be easily modified to fit more RAM banks.

Even though the processor worked on X0, 2MB of RAM was too little for some of the applications as explained in later chapters, so it was decided to also implement the processor on either X1 or X2. Both the X1 and X2 FPGAs are identical and have four

1MB RAM banks attached. For no special reason X2 was chosen and the wrapper code was modified to couple the RAM. X2 does not have direct access to the FIFOs, so the SLAAC1-V Ring bus was used and the signals were connected directly to the FIFO's inside X0. This configuration was used for all the testing and experiments in this work.

3.2 Console access

By default, the LEON processor can only communicate to the outside world through one of its two UARTs. While this might be a suitable approach for a stand-alone system, it does not make sense to connect a card in a PCI slot to the serial port in the same host computer [6]. Furthermore, the built-in UARTs of the LEON processor will be used for other purposes in the Secure Hardware Project [3]. An initial approach was to communicate through the RAM, but the handshake protocol, especially polling for new data, slowed down the programs on the LEON processor because the host has to stop the FPGA clock before it accesses the memories. Another possibility was to communicate through user registers in the PCI interface of X0. This possibility had, however, the disadvantage that the PCI interface and the user design are clocked by different signals at different rates, making synchronization very difficult. The FIFOs were chosen to transmit console data from the host to the LEON processor.

The FIFO serial interface module shown in Figure 4 consists of several tri-state buffers and a 29-bit decoder used to decode the memory controller's address bus. This decoder activates the tri-state buffers when a specific hard-coded memory address is requested in such a way that the contents of the FIFOs are present in the memory

controller's data bus. The data signals are memory mapped to 0x20000000 and the FIFO_EMPTY and FIFO_FULL handshake signals are memory mapped to 0x20000004. FIFO A0 is used to transmit data from the host to the LEON and FIFO B0 is used to transmit data from the processor to the host. The transmissions are done one character at a time because these FIFOs are 1-word deep and only eight bits of the 64-bit word are being used. The host constantly polls for new data on FIFO B0 and sends console input to FIFO A0. A state machine in the wrapper controls the FIFO_READ and FIFO_WRITE lines. A simple I/O library with *open()*, *read()* and *write()* functions was developed to support this FIFO interface from LEON processor programs.

Chapter 4

uClinux port to LEON

As explained in Chapter 2, uClinux is a derivative of the Linux 2.0 kernel intended for processors without a Memory Management Unit (MMU) such as the LEON processor. While there are many fully operational operating systems for these types of devices, uClinux has the great advantage of the familiar Linux API. Most single-threaded applications that run on the usual distributions of Linux can run under uClinux by simply linking the object code with the uClinux libraries. Multithreaded applications require some modification because of the inherent features of a MMU.

Jeff Dionne of Lineo Inc. started the project to port uClinux to the LEON processor in September 2000 [12]. Because having such an operating system running on the LEON processor would simplify the Java Virtual Machine port, it was decided that this work should contribute to the porting process in its early stages. After months of debugging and collaborative work with developers from all over the world, it was possible to run threads, mount the root filesystem, and successfully load and use simple drivers on the LEON processor. Even though the kernel development has been finished, the port cannot be considered complete because it is not possible to build executables for the user space.

In order to build executables two pieces of software are yet to be ported: UC-LIBC and the flat binary format tool chain.

UC-LIBC is a streamlined C-library specifically designed to be comprehensive but small enough for embedded devices [12]. A C-library is always platform dependent, so plenty of assembly code needs to be written for implementation on the LEON processor. Other libraries can be used such as NEWLIB and GLIBC that provide more functionality, but add up to 200k to every program. There have been initial attempts to port UC-LIBC; however, it is not possible to test them without the flat toolchain.

uClinux native executable format is the flat binary format, which is a relatively simple binary format, intended solely to contain the bare minimum needed to load and execute simple binaries [12]. A special linker script is required to prepare an ELF file, and then a special utility called *elf2flat* needs to be ported. Unfortunately, both of these pieces of software are not easy to port because they deal with reallocation of sections, which are platform dependent.

Despite not being able to generate executables, it is still possible to run applications by linking them into the kernel and using the system calls directly. Obviously, the application will have all of the kernel's privileges, but because there is no memory protection under uClinux, it does not make any difference.

The sections describe some of the major contributions of this work to the uClinux/LEON project (Sections 4.1, 4.2 and 4.3) as well as the modifications and additions specific to this work (Sections 4.4 and 4.5).

4.1 Trap handling

A trap in a SPARC processor is a vectored transfer of control to the supervisor software. Traps can be caused by an instruction-induced exception, such as a divide-by-zero or an external interrupt, such as a reset signal [38]. The transfer of control is through a special trap table that contains the first four instructions of the trap handler. Modifying the Trap Base Register (TBR) in the processor's Integer Unit can modify the base address of the table at runtime [38].

The internal boot ROM of the LEON was defined at synthesis time to immediately jump to 0x40000000, the first memory location of RAM. This was done so that all of the initialization code can run without having to synthesize and compile the design in the FPGA for small changes. Additionally, the first entry in the trap table is the reset, so it was decided to place base address of the trap table at the beginning of RAM.

The trap table was created by merging the trap tables from the RTEMS/LEON distribution and the Linux/SPARC distribution. Not all of the entries in the table have independent handlers; the only handlers implemented were reset, window overflow, window underflow, flush window, and Linux syscall. A default handler BAD_TRAP

handled the remaining traps. For debugging purposes, BAD_TRAP displays to the console the trap number and the contents of the Integer Unit registers.

The reset trap is primarily the code from locore1.S. The code initializes different registers of the processor, such as the memory configuration registers, time scaler register, and UART scaler register; finally, it enables the traps and jumps to the Linux kernel. The window overflow and underflow traps are discussed in Chapter 2. The flush window and the Linux syscall traps were directly copied from the Linux/SPARC distribution and basically call assembly language functions that flush the windows and call the appropriate functions, respectively.

4.2 Window overflow/underflow management

The SPARC Architecture includes the concept of "register windows," a unique feature designed to reduce function call overhead [38]. At a given time the instructions can access eight global registers and a window of twenty-four registers. This register window comprises the eight "in" and eight "local" registers together with eight "out" registers that will become the "in" registers of the next function called. The SPARC architecture does specify a minimum of two and a maximum of thirty-two register windows for an implementation; the LEON processor has eight.

The limitation of a fixed number of register windows does not present a problem for small programs that do not have deep function calls. However, bigger programs and operating systems will have windows that overlap. For instance, on an eight register

window processor the ninth function call will try to use the window from the first function. When such an overlap occurs, the Integer Unit generates a window overflow trap that needs to be handled by the supervisor software. The window overflow handler usually saves the contents of the register to memory, which later gets restored by the window underflow handler

For uClinux, the window overflow and underflow handlers were originally copied from the Linux/SPARC distribution. That version was further simplified because it used the MMU to save and restore the register window into the user space of each process. Debugging was long and tedious because optimizing compilers make it difficult to predict when the traps will happen. Furthermore, incorrect behavior of the underflow and overflow handlers usually gives errors that are obscure and difficult to trace.

4.3 Root filesystem implementation

The root filesystem is the device that is mounted first as the directory called "/". Any other device can only be mounted as a subdirectory inside the root filesystem. Because uClinux supports all of the filesystem types supported by the Linux 2. 0. 14 kernel, it was not a trivial decision which type of filesystem to implement on the LEON processor. The first idea that came to mind was to mount the root filesystem through NFS, transmitting the data via the FIFOs or one of the buses in the SLAAC1-V board. However, developing the driver was not an easy task and there were additional options that could be explored for implementation.

Another alternative was to use the Extended Filesystem 2 (EXT2FS). This type of filesystem is a very flexible and efficient filesystem specifically designed for large partitions and has become Linux's native filesystem [39]. Generating a file that contained the filesystem required the creation of a loopback device that was mounted on the host computer. The files needed in the root filesystem of the LEON processor were then copied to the mounted subdirectory. Once the file containing the LEON root filesystem was generated, it was copied to a specific memory location in one of X2's memory banks and its location was hard-coded into the kernel. The EXT2 filesystem worked satisfactorily; however, it required knowledge of the total size of the filesystem at compile-time and its overhead was excessive for small filesystems.

Finally, it was decided to use the ROM Filesystem (ROMFS) even though RAM would be wasted because it would only be used as ROM. The ROM Filesystem is a small read-only filesystem with very little overhead originally designed for use in recovery disks [40]. It operates in block devices and its very simple underlying structure will be explained in Section 4.5. To generate the filesystem, a user program called *genromfs* was required. This program takes a directory on the host computer and generates a file that contains all of the directory's files and subdirectories. This generated file is then converted to S-record format and linked into the kernel (see Section 4.4). Because the filesystem was included into the kernel at link-time, it was possible to associate variables with its location and size, avoiding hard-coded absolute addresses in the kernel.

This ROM Filesystem implementation of the root filesystem worked satisfactorily so it was used for all of the tests and experiments in this work and is now being used in the public distribution of the uClinux/LEON port.

4.4 Memory map

This Chapter explains the organization of different sections of memory to adapt to the specific characteristics of the SLAAC1-V board. Figure 5 shows a simplified memory map (not to scale). Note that even though there are four independent memory banks in X2, uClinux accesses all 4MB of RAM as one flat memory space due to the flexibility of the LEON processor's memory controller and wrapper code explained in Chapter 3.

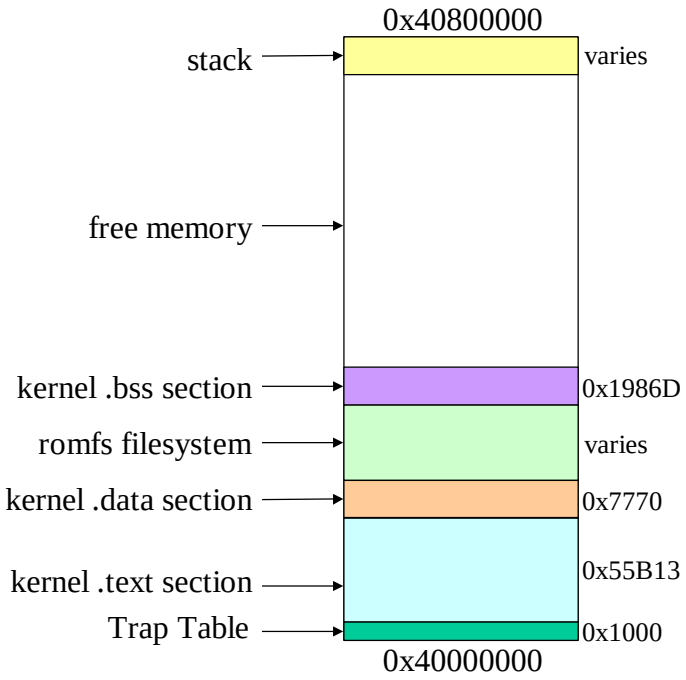

Figure 5 uClinux memory map

As stated in Section 4.1, the trap table is placed at the beginning of the first RAM bank to take advantage of the fact that the first entry in the table is the reset trap

instruction. The linker script was modified so that the linker joins all of the .text, .data and .bss sections independently and places them according to Figure 5. The ROM filesystem was placed before the .bss section intentionally to keep all of the read-only sections adjacent. In the future, when ROM is available, all of the sections below .bss can be placed in ROM by changing only a couple of lines in the linker script.

For some experiments it was necessary to modify the memory map detailed in this chapter because the current version of uClinux cannot dynamically allocate more than 1MB of consecutive memory. As will be seen in Chapter 6, some experiments require more than 1MB for Waba's object heap. Therefore, the memory map was modified so that uClinux manages a smaller amount of memory, and Waba uses a hard-coded area that is not managed by uClinux for its class and object heaps.

4.5 Direct access to the ROM Filesystem

While trying to run JBits programs under Waba on the LEON, it was found that the kernel was quickly running out of memory. Further investigation showed that whenever Waba needs to load a class, it dynamically allocates memory for the full length of the class file and then uses *read()* to copy the entire file to the allocated memory. Internally, the kernel *read()* function also allocates memory, copies the data to it using the romfs driver, and then transfers the data to the memory allocated by Waba. When using slow permanent storage and when there is a distinction between kernel-space and user-space, it makes sense for the kernel to buffer the data and then copy it to user-space. However, in the implementation for this work, three redundant copies are completely unnecessary and

copying the entire file from one memory location to another significantly slows down the program. Furthermore, the original data is already in RAM because the class files are inside the ROM filesystem. As stated in Section 4.3, the ROM filesystem has a very simple structure and it can be exploited to provide a direct access to the files without kernel buffering. Figure 5 shows the layout of the filesystem [40]:

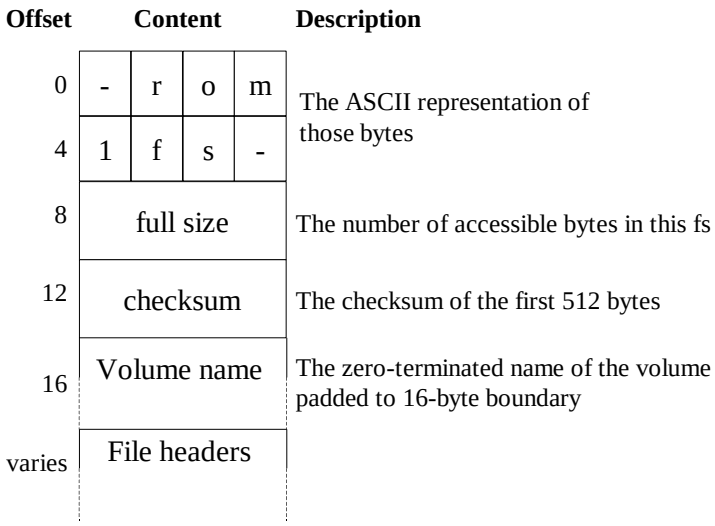

Figure 6 ROM Filesystem layout

The simple structure of the filesystem makes it easy to find the first file headers, which in turn have simple structure as well, as Figure 7 shows.

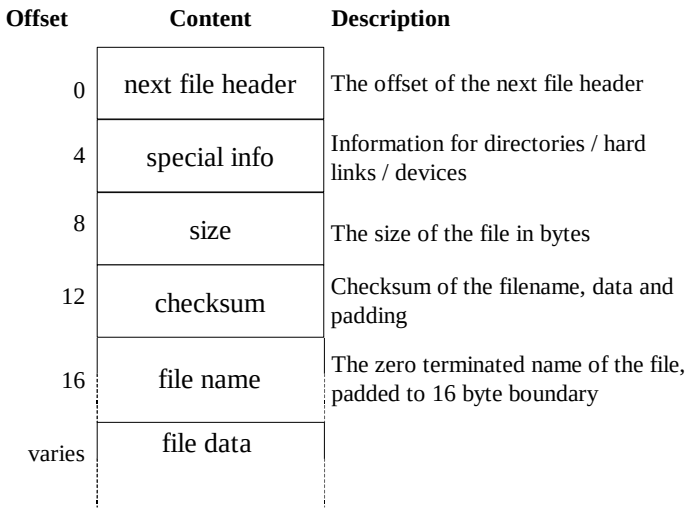

Figure 7 ROM Filesystem header layout

Taking advantage of the simple structure of the headers, a function that performs a linear search on each file header was written. The function's input is the name of the file and it returns a pointer to the absolute memory location of the file's data. The function was then included in the ROM Filesystem driver as the *lseek()* file operation. Obviously, the function does not perform the expected *lseek()* operation, but it does return an integer that can be cast to a pointer.

By using this modification to the ROM Filesystem, any user program can know the absolute address of a file in the root filesystem. While this feature violates the encapsulation and information hiding software-engineering principles, it saves a significant amount of RAM, which is always a limitation in embedded systems.

Chapter 5

Java Virtual Machine port

This chapter explains the implementation of Waba, the Java Virtual Machine used in this work, to the SLAAC1-V board running uClinux. Chapter 2 explained the main features of Waba, so this chapter will focus primarily on the development of ports of Waba to x86/Linux (Section 5.1) and SPARC/uClinux (Section 5.2). This chapter also covers the development of the Java API classes not included in the Waba Foundation classes, but required to execute JBits programs (Section 5.3). Additionally, Section 5.4 explains in detail the development strategy to be followed to execute JBits or any console-based Java program on the FPGA platform.

5.1 Waba port to x86/Linux

Because Linux and uClinux share a common API, it made sense to develop a port of Waba to an x86/Linux computer before venturing into developing the port for a SPARC/uClinux computer. This was especially important because at the time the Waba port started, the uClinux kernel was still under development. Debugging a user program in an unreliable kernel would have been very time consuming.

Waba was designed to be very portable, so the core of the Virtual Machine did not need any modifications from the PalmOS port. However, some data types were redefined in the header files so that the size of the elementary data types that Waba uses match the ones of an x86 computer. Conversion functions for these data types had to be included as well.

The Waba core expects all ports to include a few functions that can only be handled natively [17]:

- Some sort of *main()* function that is called by the operating system, clears the heaps, sets up the location of the classes and calls the core's VmInit with the appropriate parameters.
- A *loadFromMem()* function that opens a class file, reads its contents and returns a pointer to the first byte of the class file.
- A *getTimeStamp()* funtion
- A *ui_init()* function, to initialize the User Interface.
- A *ui_exit()* function, to exit from the User Interface.
- A *drawErrorWin()* function that draws an error window in a Graphical User Interface.
- A *drawMainWin()* function that draws the program window for programs that use the Graphical User Interface.
- A *handleMainWinEvent()* function that handles events such as *penMove*, *penUp*, and *penDown*.

Because the final platform for this work does not have a graphical user interface, the last five functions were left empty. The remaining functions were based on the code from the PalmOS port and were not difficult to write.

Once the Waba core compiled and linked successfully, the next step was to develop the native functions called by the Waba Foundation classes. Most of these native functions deal with the graphical user interface (see Chapter 2), so they were not ported. The native functions ported are the ones from the *File*, *SerialPort*, *Convert*, and *Vector* classes. Once again, the code was based on the PalmOS port except for the *File* class. This class was developed from scratch because PalmOS does not have a filesystem per se.

Because Waba was originally designed for mobile PDA-like devices where all of the user interaction is done through a GUI, the Waba Foundation classes do not include support for a standard I/O or text console. A text console, however, is the only interaction device on the LEON. Therefore, it was decided to implement a class that resembles the System class of the Java API. Unfortunately, this class could not be placed as part of the *java.lang* package as in the Java API because the Java compiler uses the *System* class from the host, which generated a name conflict. This name conflict could not be resolved, so it was decided to place the *System* class in the *waba.io* package. This placement, however, has the disadvantage that all the Java source code needs an "import waba.io.System" line for the program to be able to use the text console.

Weeks after the port of Waba explained in this chapter was finished, a group of open-source programmers released a complete port of Waba to x86/Linux. This latter port includes GUI support through GTK libraries as well as a port of all the Waba Foundation classes [17].

5.2 Waba port to SPARC/uClinux

Once the Waba port to x86/Linux worked satisfactorily, it was a minor task to compile the source code with the SPARC cross compiler. Nevertheless there were some issues that needed to be addressed:

- SPARC is a big-endian architecture as opposed to the little-endian x86 architecture, so the data type conversion macros had to be modified.
- Because Waba will be running as part of the kernel and not as a user program, the *main()* function of Waba was renamed to *mainWaba()* and is called from the kernel after all the initializations are completed.
- For the same reason as above, the kernel functions *kmalloc()*, *kfree()*, and *printk()* had to be used instead of the user functions *malloc()*, *free()*, and *printf()*.
- Some functions from the standard C and Math libraries had to be written because the kernel did not have equivalent functions. Examples are *atoi()*, *itoa()*, *pow()*, and *exit()*.
- To save RAM, the direct ROM Filesystem access described in Chapter 4.5 had to be used in *loadFromMem()*.

Once these modifications were made, the SPARC/uClinux port of Waba worked exactly like the x86/Linux port.

5.3 Java API classes implementation for Waba

As stated in Chapter 2, all Java programs can assume that the complete Java API will be available with the Java Virtual Machine. However, the Java API contains more than 570 classes with each one having several methods [41]. The majority of those classes and methods will not be used by JBits programs; so the only classes and methods ported were the ones required by the programs in the experiments of Chapter 6. The following sections describe these classes organized by Java API packages.

5.3.1 java.lang

The *java.lang* package provides fundamental classes to the design of the Java Programming Language [41]. The classes ported in the *java.lang* package were *Character*, *Integer*, and *Class*.

The Character class provides Java programs with a wrapper of the primitive type "char" in an object. The class provides methods for converting characters from uppercase to lowercase and vice versa [41]. The JBits API uses this class when determining the type of device because all names are maintained internally as uppercase, but the parameters provided by the user can be either uppercase or lowercase. The algorithms used in the methods were mostly copied from KAFFE, an open-source Java Virtual Machine and Java API implementation [42].

The Integer class provides Java programs with a wrapper of the primitive type “int” in an object. The class provides methods for converting an integer to other data types, such as a String, and other data types to an integer. The JBits API does not use this class directly, but it is very useful when parsing command-line arguments of programs. Once again, the algorithms and even some methods were a verbatim copy from KAFFE.

The *Class* class provides a representation of classes running in a Java application. The class provides methods to access the name of the class, the name of the superclass, the name of the package, the names of the methods, and other properties such as if the object is an interface or a primitive. Some methods in this class can also create new instances given a string of the class name. These latter methods are used by the JBits API to load internal classes based on the type of device of the bitstream. Considering that the *Class* class methods are native methods that need to access the Java Virtual Machine class loader, it was not possible to copy the source code from KAFFE. Thus, the class and methods were developed from scratch.

5.3.2 java.util

The *java.util* package contains collections of utility classes for data structures, date and time facilities, internationalization, string parsing, random number generator, and bit arrays [41]. The classes ported in the *java.util* package were *Stack*, *StringTokenizer*, *Calendar*, and *GregorianCalendar*.

The *Stack* class represents a last-in-first-out collection of objects. It extends the *Vector* class to allow the collections to be treated as a stack. The methods provided are *push()*, *pop()*, and *peek()* along with a method to test if the stack is empty and a *search()* method that returns how far an object in the stack is from the top. JBits uses this class extensively to organize the different data structures found in the bitfiles. The class and its methods were developed from scratch and it was a trivial programming exercise because all the memory management was performed by the superclass *Vector*.

The *StringTokenizer* class allows a Java program to break a string into tokens. The class provides methods to count the number of tokens, get the next token according to a delimiter, and query if there are more tokens in the string. The JBits API does not use this class, but the JBits program in Section 6.2 uses the *StringTokenizer* class to parse a file that contains new data. The class and its methods were adapted from the distribution of KAFFE [42].

The Calendar and *GregorianCalendar* classes provide methods to obtain the current date of the System. These classes are used by JBits to set the date field in the generation of a new bitstream. Because the LEON does not have a real-time clock, the methods in these classes always return a fixed date.

5.3.3 java.io

The *java.io* package provides system input and output through data streams and the local filesystem. The classes ported in the *java.io* package were *File*,

BufferedInputStream, and *BufferedOutputStream*. The *SerialPort* class is not a part of the *java.io* package according to the specification but it was included to support the UARTs of the LEON.

The *File* class interfaces Java programs to the operating system's filesystem by providing methods to open, close, read and write files. JBits does not use this class directly, but it uses BufferedFileInputStream and BufferedFileOutputStream classes that contain objects of type File. The class and methods were developed from scratch because the methods are mostly native and require knowledge of the Java Virtual Machine internals. Nevertheless, the development of the methods was a simple task thanks to the Linux API.

The *BufferedInputStream* and *BufferedOutputStream* are classes that represent a stream of data from or to a file in the filesystem. The class methods include *read()*, *write()* and *skip()*, while the *open()* function is implied in the constructor. These classes are used by the JBits API to read and write the bitfiles to the filesystem. The methods were easy to develop once the *File* class was completely debugged. Even though the name of these classes imply that the contents of the file streams are temporarily buffered, the implementation developed for this work does not buffer the data to save valuable memory.

5.4 The development strategy

While attempting to run the JBits programs under the SPARC/uClinux port of Waba, it was found that it is much easier to develop the applications in the fast x86/Linux port. This finding suggested a strategy for testing and developing Java, and especially JBits applications for the FPGA platform. This chapter explains this four-step testing and development strategy. See Section 6.1 for a working example.

Step one of the suggested strategy is to write and test the program on any platform that has a certified Java Virtual Machine and a complete Java API implementation. For this work, SUN'S JAVA RUNTIME ENVIRONMENT 1.3.1 was used [43]. Starting with this step has the advantage that it can be assumed that any unexpected behavior of the program will be due to a mistake in the JBits program, rather than a bug in the hardware, operating system, Java Virtual Machine, or Java API implementation.

Step two is to add the necessary Waba imports to the source code file and try to run the program under the x86/Linux port of Waba. This step has the feature that unexpected behavior will be due to a bug in the Waba port or Java API implementation, not due to a bug in the hardware, operating system or JBits program.

Waba imports are needed whenever the package of the used classes differs from the package in the Java API specification. A specific example is the *System* class that was moved to the *waba.io* package because of the reasons explained in Chapter 5.1. Consequently, it is necessary to add an "import waba.io.System" to the source code.

In this step it is possible to obtain the amount of memory in Waba's object and class heaps required to execute the JBits program. The sizes of these heaps are command line arguments and the minimum required can be obtained by trial and error. Moreover, this step is where the classes of the Java API that are not implemented in Waba will result in errors such as:

```
Error: can't find class java/util/Stack in
santi/J/FSMROMCfg. main([Ljava/lang/String;)V
Please notify the program's author.
*** EXITTING ***
```

A similar error is displayed if a method is not implemented. The programmer then has to port the class and/or the methods and copy the compiled bytecodes to the Waba directory. The behavior of the methods is thoroughly explained in the Java API specification [41]. The libraries from KAFFE [42] can be used as a starting point.

The third step of the suggested strategy is to run the program on the SPARC simulator. For this step, it is necessary to link the uClinux kernel together with a filesystem that contains the directories of all the class files. This step will find bugs in the SPARC/uClinux version of Waba or in uClinux itself because it can be assumed that the hardware, JBits program, and Java API implementations have already been tested in previous steps.

Finally, the fourth step is to run the program on the SLAAC1-V board. For this step, it is only necessary to make sure that the program can fit in memory and make the

necessary adjustments to the memory map as explained in Chapter 4.4. This step tests the hardware and memory organization exclusively and is the conclusive test for the program.

Chapter 6

Applications and Experiments

This chapter shows some experiments that were developed and tested using the recommended strategy. The purpose of the experiments is to demonstrate that it is possible to run console-based Java programs and specifically JBits applications in a FPGA. Additionally, the chapter analyzes the memory requirements, performance measurements, and performance bottlenecks of the JBits applications.

The x86/Linux versions of the programs, as well as the SPARC simulator, were executed on a Pentium-II 300MHz with 256MB of RAM running Linux 2.2.19. The running time for different parts of the program were obtained by sending debugging strings within the program to the standard output. This output was piped through *tai64n* and *tai64nlocal* from the DAEMONTOOLS package, which put a precise timestamp on every line of the standard output [44]. For example:

```
2001-05-25 23:42:14. 687720500 Writing out bitfile
2001-05-25 23:43:05. 580050500 Done writing!
```

6.1 Simple Java Application

The main objective of this experiment is to familiarize the reader with the four-step strategy by applying it to the development of a very simple Java application. This simple application also tests the functionality of the Java Virtual Machine in such key aspects as loading classes and executing different types of methods (class, static and native), as well as testing the API classes ported to this implementation of the Java Virtual Machine.

This is the source code for a simple program that outputs "Hello World" to the standard output:

```
public class hello
{
    public static void main(String  args[])
    {
        System. out. println("Hello World!\n");
    }
}
```

Step 1: Compile it and run it using SUN'S RUNTIME ENVIRONMENT (version 1.3.1 was used throughout these experiments)

```
$ javac hello. java
$ java hello
Hello World!
```

Step 2: Implement the necessary Java API classes for Waba, add the necessary imports and run the program using Waba for the Linux/x86 platform:

```
$ waba -a hello4
    class:     [hello]
    classpath: [hello:/usr/local/java/jdk1-118sun/lib/classes.
  zip:/home/sleon/jvm/waba/waba_classes:/project/vtloki/JBits:. :]
    vmStackSize    = 1500
```

```
nmStackSize    = 300
classHeapSize  = 14000
objectHeapSize = 8000

startApp(): calling VMInit
startApp(): calling VmStartApplication

Hello World!
```

Step 3: Compile the kernel and run the program in simulation:

```
$ make linux srec
$ sis64 linux.srec
SIS - SPARC instruction simulator 3. 1. 0,  copyright Jiri
Gaisler 1995-1998
. . .
(output trimmed)
. . .
Hello World!
```

Step 4: Run the program on the SLAAC1-V board:

```
$ make bin
$ leonx2
```

Output on the serial port:

```
uClinux/Sparc
Flat model support (C) 1998-2000 Kenneth Albanowski, D.  Jeff
Dionne
LEON-2. 1 Sparc V8 support (C) 2000 D.  Jeff Dionne, Lineo
Inc. , Santiago Leon, Ruediger Jordan, Virginia Tech
Configurable Computing Lab
. . .
(output trimmed)
. . .
Hello World!
```

Because this is not an actual application, but rather a small program used to demonstrate the design strategy, neither performance measurements nor memory requirements were obtained. However, it is important to note that once the program runs under a traditional Java runtime environment such as Sun's JRE, the source code does not

need any modifications for it to run under the Waba platforms (with the exception of the additional "import" lines). Furthermore, once the compiled Java code runs on the Linux/x86 Waba, the same code can be used for the simulation and for the hardware steps.

6.2 JBits template lookup and LUT modification for a Xilinx 4085 part

In the previous experiment it was shown that designs can be tested on the fast and flexible Linux/x86 Waba and then use the same class files for simulation and hardware. This experiment will show that this design and testing strategy can be used to execute a more complex JBits code.

This program modifies the transition table of a state machine synthesized in a FPGA. It was originally written by Ryan Fong of the Configurable Computing Lab at Virginia Tech to change the state machine that switches the contexts of an integrated circuit in the Sanders' CSRC board [45]. First, the program opens a text file that specifies the transition table. The format of this file is a ROM configuration format used in Xilinx's CoreGen. Second, it proceeds to search into each CLB of the original bitfile for a vector that will be replaced by the contents of the new transition table. Finally, the program writes the modified bitfile to memory.

The program appears to be simple; however, the search for the original template is not trivial. The placing directives in the hardware description language are only relative, so there is no way of knowing the absolute placement of the column of CLBs to be modified. Furthermore, the compilation tools of the FPGA vendor scramble the address

and LUT vector configurations [19]. Jonathan Ballagh solved these problems by making sure that the original "vector was unique in the sense that every possible address configuration yielded a unique output sequence from the LUT" [19]. Thus, the search becomes a linear search of all the CLBs, looking for the distinctive vector in any of its scrambled forms.

Table 2 shows the memory usage for this experiment. Note that the components under "Compiled Source" and "Filesystem" can reside in ROM. However, the SLAAC1-V board does not contain any ROM, so those components were placed in RAM.

	Components	Size (bytes)
Compiled source	uClinux	386,496
	Waba	58,880
Filesystem	Program classes	12,928
	Waba classes	155,648
	JBits classes	942,080
	Original Bitstream	240,711
Waba runtime	Object Heap	950,000
	Class Heap	190,000
	JVM Stack	1,500
	Native methods Stack	300
Total		**2,938,543**

Table 2 Memory usage of template lookup and LUT modification for a Xilinx 4085 part

Table 3 summarizes average running time of five iterations for different parts of the program (all units are seconds):

	Sun JDK 1.18		Waba		
	x86/linux		x86/linux	sparc/uClinux	
	with JIT	without JIT		simulation	hardware
OS Loading	N/A	N/A	N/A	36.480	2.600
JVM initialization	0.179	0.170	0.310	39.745	2.130
Initialization of JBits classes	0.451	0.516	4.697	1741.692	122.460
Reading the Bitstream	0.269	0.270	11.610	8359.251	662.550
Bitstream Search	1.295	1.417	45.530	21751.736	2072.300
Bitstream Modification	0.007	0.008	0.169	76.220	7.260
Writing the Bitstream	0.031	0.044	0.450	2925.730	337.905
Total	**2.232**	**2.425**	**62.766**	**34894.374**	**3204.605**

Table 3 Results of template lookup and LUT modification of a Xilinx 4085 part

The table shows that meanwhile the hardware is many times faster than the simulation, the x86/Linux Waba is almost 60 times faster than the application running on the FPGA. This ratio emphasizes the importance of developing and testing the applications on the x86/Linux version of Waba first. Furthermore, the x86/Linux version is much more flexible because changes in the code only require a Java compilation, while the hardware needs additional compilations of the filesystem and the uClinux kernel.

Table 3 also shows that more than 80% of the total time it takes to run the program is spent on initialization. Bitstream search is considered an initialization because the original template bitstream will not change. Therefore, the memory and the registers of the processor can be saved into a snapshot. Then, whenever the program needs to run, the initialization snapshot can be loaded back, which will save much processing time.

Another performance bottleneck when running this program is the amount of time that the processor is writing back the bitstream to memory. This is an inherent problem of the current versions of JBits because it was not designed with embedded systems in mind. A Xilinx 4085 part has a bitfile of 240KB, and every single byte is written

independently by calling a *write()* function in a *BufferedOutputStream* class. Thus, for every byte there is an expensive overhead of a Java function call. This explains the difference between the JIT and no-JIT runs when writing back the bitstream, because a non-JIT function call is much more expensive that a JIT function call. To make matters worse, the tested implementation of *write()* under Waba contains a native function, which implies an additional overhead of a native function call. For a fast computer with multiple cache levels the function overheads might not be relevant, but for a smaller processor with a single level of cache, the function overheads directly affect the performance of an application.

6.3 JBits LUT modification for Virtex 200

The previous experiment showed a real application for the FPGA platform that was used to modify a bitstream of a Xilinx 4000 series part. However, the newer, faster, and denser Virtex parts are quickly displacing the XCV4000 parts. For example, the older SLAAC1 board was populated with XCV4000 parts, while the newer SLAAC1-V board is populated with Virtex parts. Thus, it became important to develop an experiment that will modify the bitstream of a Virtex part.

The part chosen for this experiment is an XCV200 because it is one of the four Virtex parts in the SLAAC1-V board. The program is very simple: it reads the original bitfile, changes the values of LUTs in specific CLBs, and writes the bitfile back to memory. The modified CLBs were randomly chosen because the experiment's purpose was to prove that it is possible to modify a Virtex bitfile using the FPGA platform, not necessarily do

something useful with it. To verify that the program does modify the bitstream, the modified bitstream was opened in BoardScope and the individual CLB's were visually examined.

Table 4 shows the memory usage for this experiment. Note that the components under "Compiled Source" and "Filesystem" can reside in ROM.

	Components	Size (bytes)
Compiled source	uClinux	386,496
	Waba	58,880
Filesystem	Program classes	2,129
	Waba classes	155,648
	JBits classes	1,449,984
	Original Bitstream	167,058
Waba runtime	Object Heap	1,300,000
	Class Heap	320,000
	JVM Stack	1,500
	Native methods Stack	300
Total		**3,841,995**

Table 4 Memory usage of LUT modification of a Xilinx XCV200 part

The following table summarizes average running time of five iterations for different parts of the program (all units are seconds):

	Sun JDK 1.18		Waba		
	x86/linux		x86/linux	sparc/uClinux	
	with JIT	without JIT		simulation	hardware
OS Loading	N/A	N/A	N/A	36.480	2.600
JVM initialization	0.179	0.170	0.310	39.745	2.130
Initialization of JBits classes	0.640	0.651	9.501	2779.801	164.390
Reading the Bitstream	0.466	0.467	28.274	8359.251	398.020
Bitstream Modification	0.011	0.013	0.442	125.820	9.460
Writing the Bitstream	0.054	0.065	0.744	2848.913	194.570
Total	**1.350**	**1.366**	**39.271**	**14153.530**	**768.570**

Table 5 Results of LUT modification of a Xilinx XCV200 part

The table shows some the same characteristics of the previous experiment:

- The Sun JDK Virtual Machine is significantly faster than Waba under any platform.
- The x86/Linux version of Waba is roughly twenty times faster than the sparc/uClinux version running on the FPGA.
- The Sparc simulator runs nearly twenty times slower than the LEON core running on the FPGA.
- About 80% of the processing time is spent on initializations, which can be avoided by taking a snapshot of memory and processor registers, as explained earlier.
- Writing back the bitfile is another performance bottleneck because of the function overhead for every byte written to memory.

Because the program itself is not very complicated, most of the time that the program is running is spent on bitstream I/O. However, the Virtex series support partial readback and reconfiguration, which means that instead of loading and saving a complete bitfile, the program can use a partial, smaller bitfile. These partial bitstreams only contain the information of specific columns of CLBs, so reading and writing the bitstream would take a fraction of the time it currently does [18].

6.4 JBits LUT modification for Virtex 1000

One of the motivations for this work was to use it as a reconfiguration tool in the Secure Hardware Project at the Configurable Computing Lab. The first prototype of this project uses the FPGA's in the SLAAC1-V board to hide the details of the coding,

authentication, and encryption of a software radio. These components of the radio reside in either X1 or X2 FPGAs, which are Virtex 1000 parts. Therefore, it is important that the FPGA platform is able to modify Virtex 1000 bitfiles.

The program of this experiment is exactly the same used in the previous experiment with the obvious exception that JBits initializes a bitstream for a XCV1000 part. Furthermore, the modified CLB's are directly connected to LED's in a daughtercard for additional verification of the modified bitstream.

Table 6 shows the memory usage for this experiment. Once again, note that the components under "Compiled Source" and "Filesystem" can reside in ROM.

	Components	Size (bytes)
Compiled source	uClinux	386,496
	Waba	58,880
Filesystem	Program classes	2,093
	Waba classes	155,648
	JBits classes	1,527,808
	Original Bitstream	766,042
Waba runtime	Object Heap	5,680,000
	Class Heap	330,000
	JVM Stack	1,500
	Native methods Stack	300
Total		**8,908,767**

Table 6 Memory usage of LUT modification of a Xilinx XCV1000 part

The following table summarizes average running time of five iterations for different parts of the program (all units are seconds):

	Sun JDK 1.18		Waba		
	x86/linux		x86/linux	sparc/uClinux	
	with JIT	without JIT		simulation	hardware
OS Loading	N/A	N/A	N/A	36.480	N/A
JVM initialization	0.179	0.170	0.310	39.745	N/A
Initialization of JBits classes	1.941	2.254	14.159	4140.479	N/A
Reading the Bitstream	1.899	1.838	96.208	19777.221	N/A
Bitstream Modification	0.069	0.084	0.443	125.760	N/A
Writing the Bitstream	0.133	0.142	0.954	12839.462	N/A
Total	**4.221**	**4.488**	**112.074**	**36922.667**	**0.000**

Table 7 Results of LUT modification of a Xilinx XCV1000 part

Note that the table doesn't show values for the sparc/uClinux hardware execution of the program. As shown in Table 7, the memory required to run this application was much more than the available memory for the FPGA. While an XCV200 requires a total of 1.6MB for the class and object heaps, an XCV1000 requires almost 6MB. As stated before, X2 in the SLAAC1-V can only access up to 4MB of RAM, which made it impossible to run this experiment on this FPGA.

Even though a XCV1000 has nearly five times as many configurable components than a XCV200, the table shows the same trends and performance bottlenecks that were analyzed in the previous experiment.

Chapter 7

Conclusions

7.1 Summary

This thesis introduced a FPGA platform that can be used to execute JBits programs that modify various programmable resources of a FPGA. The platform required development of ports of a processor, an operating system, a Java Virtual Machine, and a set of Java API classes. The thesis also presented a developing and testing strategy for such programs.

Following the introduction in Chapter 1, Chapter 2 overviewed the topics that form the basis of this work: the SLAAC1-V board, the LEON processor, the uClinux operating system, Java, and the JBits API. Chapter 3 described the steps that were necessary to implement the LEON processor to the SLAAC1-V board with special attention to access to memory and I/O. Chapter 4 explained the contributions of this work to the LEON port of the uClinux operating system. The chapter also explained some specific optimizations to uClinux developed for this work. Chapter 5 described the ports of Waba to Linux running on a standard PC and the port of Waba to uClinux running on the SLAAC1-V board. Both ports required the development of Java API classes that were described in the chapter as well. The chapter also explained in detail each step of the suggested

development strategy including what aspects that are tested at each step. Chapter 6 presented some programs running on the platform that modified the configuration of FPGAs along with performance and memory measurements.

7.2 Suggestions for Future Work

This research creates several opportunities for future investigations. First, there are several improvements that can reduce the time required to execute JBits applications on the platform:

- implementing a snapshot of the memory and processor registers after the initialization of the JBits classes, as explained in Chapter 6,
- increasing the amount of memory so that Waba's garbage collector runs less often,
- using partial bitstreams for Virtex parts, and
- rewriting some JBits classes to avoid the overhead of reading and writing bitstreams one byte at a time.

Second, this work can be used as part of a framework for remote reconfiguration of FPGAs. The framework would utilize the transmission link more efficiently because a class file of a JBits program will be significantly smaller than an entire bitfile. Further, it might not be necessary to transmit a program, but rather only its parameters. Security would another important advantage of the framework. If an entire bitstream is transmitted over a medium, this medium can be intercepted and the bitstream can be obtained, revealing the inner details of the design. Alternatively, if a JBits program is

intercepted, it may be useless without the original bitfile. Furthermore, because the framework runs on a FPGA, it can be secured using schemes such as those used in the Secure Hardware Project at Virginia Tech's Configurable Computing Laboratory [3].

7.3 Conclusions based on the Work

The objective of this thesis was to demonstrate that it is possible to modify the configuration of a FPGA within a FPGA. The JBits API provided a set of classes and methods that enabled reconfiguration of FPGAs in significantly less time than is required when using traditional synthesis and compilation processes. However, the execution of JBits programs presented another challenge, the implementation of a Java Virtual Machine on a FPGA board. The underlying platform required for a Java Virtual Machine was implemented using generic processor core and a Unix-like operating system. Once the underlying platform executed satisfactorily, a Java Virtual Machine was ported along with several Java API classes. Finally, it was possible to successfully execute JBits programs on the FPGA platform, which warrants further investigation and development of applications.

Compared to the other self-reconfigurable platforms overviewed in Chapter 2, this work has the advantage that any parameter of any design in a FPGA can be reconfigured, not only a specified set of parameters. This flexibility, however, comes with the disadvantage that the reconfigurations take a significant amount of time, making run-time reconfiguration inefficient. Nevertheless, the platform has potential for execution speed improvement making run-time reconfigurations a future possibility.

Bibliography

[1] David I. Lehn, Rhett D. Hudson and Peter M. Athanas, *"Framework for architecture-independent run-time reconfigurable applications,"* SPIE Proceedings, Nov 2000.

[2] Steve Guccione, Delon Levi, and Prasama Sundararajan, *"JBits: Java based interface for reconfigurable computing."* Second Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD'99), The Johns Hopkins University, Laurel, Maryland, Sep 1999.

[3] Scott Harper, *"Secure Computational Hardware,"* Internal Documentation.

[4] Information Sciences Institute – East, SLAAC Project Page, http://www.east.isi.edu/projects/SLAAC, June 2001.

[5] *"SLAAC1-V User VHDL Guide,"* Release 0. 3. 1, Information Sciences Institute – East, SLAAC1-V software distribution, June 2001.

[6] Rüdiger Jordan, *"Radio management using an embedded RISC processor,"* Diplomarbeit of Darmstadt University of Technology and Virginia Tech, January 2001.

[7] Jiri Gaisler, *"LEON SPARC Processor,"* http://www.gaisler.com/leonmain.html, June 2001.

[8] GCC Homepage, http://www.gnu.org/software/gcc/gcc.html, June 2001.

[9] GNU Binutils, http://www.gnu.org/software/binutils/binutils.html, June 2001.

[10] RTEMS Homepage, http://www.rtems.com/RTEMS/rtems.html, June 2001.

[11] TSIM SPARC simulator, http://www.gaisler.com/tsim.html, June 2001.

[12] uClinux Homepage, http://www.uclinux.org/, June 2001.

[13] Bill Venners, *"Inside the Java Virtual Machine,"* McGraw-Hill, 1998.

[14] Ed. Paula Ferguson, *"Java in a Nutshell,"* O'Reilly and Associates, 1997.

[15] Joseph L. Weber, *"Using Java2 Platform Special Edition,"* QUE, January 1999.

[16] Tim Lindholm, Frank Yellin, *"The Java Virtual Machine Specification,"* Addison-Wesley Pub Co, April 1999.

[17] Waba Homepage, http://waba. sourceforge. net/, June 2001.

[18] Xilinx Inc. *"Virtex Series Configuration Architecture User Guide, XAPP151 (v1. 5),"* http://support.xilinx.com/xapp/xapp151.pdf, September 27, 2000.

[19] Jonathan Ballagh, *"Design of a Reconfigurable IP Content Addressable Memory Core"*, Internal Documentation, December 1999.

[20] Delon Levi and Steven A. Guccione, John Schewel, ed. , *"BoardScope: A Debug Tool for Reconfigurable Systems,"* Configurable Computing: Technology and Applications, Proc. SPIE 3526, Bellingham WA, November 1998.

[21] Altera, "Nios Embedded processor Programmer's Reference Manual," Version 1.1.1, July 2001.

[22] Tensilica Homepage. http://www.tensilica.com, June 2001.

[23] MicroBlaze Homepage, http://www.xilinx.com/ipcenter/processor_central/microblaze.htm, June 2001.

[24] OpenRISC 1000 Homepage, http://www.opencores.org/cores/or1k/, June 2001.

[25] RTLinux Homepage, http://www.rtlinux.org/, June 2001.

[26] LinuxDevices Homepage, http://www.linuxdevices.com/, June 2001.

[27] PicoJava Microprocessor Cores, http://www.sun.com/microelectronics/picoJava/, June 2001.

[28] Sun Microsystems, *"picoJava ™ -II Data Sheet,"* April 1999.

[29] A. Kim and J.M. Chang, *"Designing A Java Microprocessor Core Using FPGA Technology,"* Proceedings of 1998 IEEE International ASIC Conference, Rochester, NY, Sep. 13-16, 1998.

[30] A. Kim, Y. Qian and J.M. Chang, *"Embedded Java Processor and Memory Architecture,"* Submitted for ICCD99, September 1999.

[31] Derivation Systems Inc. press release at DAC'99, *"Derivation Systems Introduces "LavaCORE", the worlds first Formally Synthesized Java Virtual Machine FPGA Core,"* http://www.derivation.com/news/pressrelease_06-21-1999.html, June 21, 1999.

[32] Derivation Systems Inc. press release at ESC-WEST 2000. "Derivation Systems Demonstrates *"LavaCORE" Configurable Java(tm) Processor FPGA Core,"* September 26, 2000.

[33] Vulcan ASIC Inc. *"Moon 1.0 Data Sheet,"* January 15th 2000.

[34] James G. Eldredge, Brad L. Hutchings, *"RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs"*, Custom Integrated Circuits Conference, pages 77-80, San Diego, California, May 1994.

[35] Tadayoshi Horita and Itsuo Takanami, *"A Built-In Self-Reconstruction Approach for Partitioned Mesh-Arrays Using Neural Algorithm,"*IEICE TRANS., Vol E-79-D, No. 8, August 1996.

[36] Masaru Fukushi, *"Self-Reconfigurable Mesh Array System on FPGA,"* Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'00).

[37] Michael J. Wirthlin and Brad L. Hutchings, *"DISC: The dynamic instruction set computer,"* Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, John Schewel, Editor, Proc. SPIE 2607, pp. 92-103 (1995).

[38] SPARC International Inc. , *"The SPARC Architecture Manual, Version 8,"* Revision SAV080SI9308. http://www.sparc.org/standards/V8.pdf, June 2001.

[39] Ext2fs Homepage, http://e2fsprogs.sourceforge.net/ext2.html, June 2001.

[40] Janos Farkas, *"ROMFS - ROM FILE SYSTEM,"* romfs.txt in the genromfs distribution, June 2001.

[41] Sun Microsystems, *"JavaTM 2 Platform, Standard Edition, v 1. 3. 1 API Specification,"* http://java.sun.com/j2se/1.3/docs/api/index.html, June 2001.

[42] Kaffe Homepage, http://www.kaffe.org, June 2001.

[43] JavaTM 2 Runtime Environment, Standard Edition http://java.sun.com/j2se/1.3/jre, July 2001.

[44] D. J. Bernstein , *"daemontools,"* http://cr.yp.to/daemontools.html, June 2001.

[45] S. M. Scalera and J. R. Vazquez, *"The Design and Implementation of a Context Switching FPGA,"* Proceedings of the Symposium on Field-Programmable Custom Computing Machines, April 1998.

Vita

Santiago Leon was born in July 1977 in Quito, the capital city of Ecuador. After graduating from Alberto Einstein High School, he worked part-time and attended Escuela Politécnica Nacional in Quito for one year. Santiago enrolled at Virginia Tech in 1996 and began pursuing a Bachelor's Degree in Computer Engineering. During his junior year at Virginia Tech, Santiago decided to continue his studies by enrolling in the 5-year BS/MS program. He took his B.S. in Computer Engineering in May 2000 and his M.S. degree in July 2001. After graduation, Santiago began employment with IBM Microelectronics in RTP, North Carolina as a performance designer for PowerPC and PowerNP processors.