

IMPLEMENTATION OF THE .NET CLR ON FPGAs

By

Srinath S

Srinivasan T

Vidyabhushan M

A Project Report submitted to the

**FACULTY OF INFORMATION AND
COMMUNICATION ENGINEERING**

in partial fulfillment of the award
of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING, GUINDY**

ANNA UNIVERSITY

CHENNAI – 600 025

APRIL 2006

CERTIFICATE

Certified that this project report titled “**Implementation of the .NET CLR on FPGAs**” is the *bona fide* work of **Srinath S. (20022190), Srinivasan T. (20022191) and Vidyabhushan M. (20022209)** who carried out the project under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertations on the basis of which a degree or award was conferred on an earlier occasion on these or any other candidates.

Dr. P. NARAYANASAMY,

Head of the department,
Department of Computer
Science and Engineering,
College of Engineering, Guindy,
Anna University,
Chennai - 600025

Dr. RANJANI

PARTHASARATHI,
Professor,
Department of Computer
Science and Engineering,
College of Engineering, Guindy,
Anna University,
Chennai - 600025

ACKNOWLEDGEMENTS

We would like to convey our gratitude to **Dr. P. Narayanasamy**, Head of the Department, Department of Computer Science and Engineering, College of Engineering, Guindy, Anna University, Chennai for providing us the opportunity and infrastructure to carry out this project.

We express our sincere gratitude to our guide **Dr. Ranjani Parthasarathi**, Professor, Department of Computer Science and Engineering, College of Engineering, Guindy, Anna University, Chennai for her invaluable support, guidance and encouragement for the successful completion of this project. Her knowledge, her attitude, her commitment and her spirit have inspired and enlightened us.

At this juncture, we fondly remember, with gratitude, our .NET guru Late **Mr. Ramasubramanian**, who initiated us into the field of .NET.

We take this opportunity to thank **Mr. Shanmugam** of our Department for extending technical support.

We would like to thank Xilinx Inc. for providing us with equipments for our project. We would also like to thank our seniors at Xilinx namely **Mr. Siva Velusamy, Mr. Navaneethan Sundaramoorthy, Mr. Vasanth Asokan and Mr. Raj Kumar Nagarajan** for extending their support throughout the project.

And last, but not the least, we wish to thank our parents and family members for bearing with us throughout the project period and for having created the opportunity to do this course in such a prestigious institution.

TABLE OF CONTENTS

CHAPTER NO	TITLE	PAGE NO
	ABSTRACT (ENGLISH)	i
	ABSTRACT (TAMIL)	ii
	LIST OF FIGURES	iii
	LIST OF ABBREVIATIONS	iv
1	INTRODUCTION	1
	1.1 Introduction to Virtual Machines	1
	1.2 Introduction to .NET	1
	1.3 .NET in Embedded Systems	2
	1.4 Proposed Scheme	3
	1.5 Introduction to FPGAs	3
	1.6 Existing techniques to improve the performance of Virtual Machines	4
	1.6.1 Methodologies adopted to improve performance of JVM.	5
	1.6.2 Differences in design issues of CLR and JVM	6
	1.7 Organization of the thesis	6
2	LITERATURE SURVEY	7
	2.1 JVM implementation FPGAs	7
	2.2 The Pico Java Processor	7
	2.3 Hardware partition in a Co - Designed JVM	8
	2.4 Register Stack Architecture in Intel Itanium Processors	8
	2.5 Plataforma .NET	9
3	DESIGN	10
	3.1 Overall Design	10
	3.1.1 Co-Design Methodology	10

	3.1.2	Components of the System	11
	3.1.2.1	MicroBlaze SCP	12
	3.1.2.2	Custom Hardware	12
	3.1.2.3	Processor and Custom Hardware Interface	13
3.2		Detailed Design	15
	3.2.1	Software Design	15
	3.2.2	Hardware Design	17
	3.2.2.1	Control Unit	18
	3.2.2.2	Hardware Stack	19
4		IMPLEMENTATION	21
	4.1	Software Implementation	21
	4.1.1	Boot Strapping	21
	4.1.2	Metadata Initialization	22
	4.1.3	Core CLR Loop	24
	4.2	Hardware Implementation	25
	4.2.1	Control Unit	26
	4.2.2	Stack Unit	27
	4.2.3	Functional Units	28
	4.3	Interface Access	29
	4.3.1	FSL Write Operation	29
	4.3.2	FSL Read Operation	30
5		RESULTS	31
	5.1	The System	31
	5.2	Hyper Terminal Settings	31
	5.3	Test Cases	31
	5.4	Profiling	37

6	CONCLUSION & FUTURE WORK	40
7	REFERENCES	41
A	APPENDIX	43
	A.1 Xilinx Embedded Development Kit (v7.1i)	43
	A.2 Xilinx Platform Studio 7.1i	43
	A.3 Xilinx Microprocessor Debugger (XMD)	45
	A.4 Tools and reference guides	47
	A.5 Internals of the .NET Architecture	49

ABSTRACT

(English)

Microsoft's .NET platform is an innovative and promising technology to achieve true interoperability between programming languages, and true portability over different hardware and operating system platforms. A .NET embedded processor has been designed on FPGAs that improves the performance of the .NET Common Language Runtime (CLR). The flexibility of FPGAs has been exploited to accelerate programs targeted to run on .NET virtual machine by considering a constricted set of the .NET instruction set architecture. The Turing complete set of instructions has been implemented in hardware. A software-hardware co-design approach has been adopted to implement the design. The conclusion is that the performance of a hardware-software co-designed methodology is better than a purely software approach.

திட்டப்பணிச்சுருக்கம்

மைக்ரோசாப்ட்-டினீ சமீப கணிணி நிரல் மேடையான டாட்நெட் பல முக்கிய கூறுகளை கொண்டுள்ளது. டாட்நெட்டை பயன்படுத்தி நிரல்மொழிகளுக்குள் இடைசெயல்பாட்டை பெறலாம். மேலும் இதை வைத்து வன்பொருள் மேடைகளுக்கும் இடையே பெயர்வுத்திறனை அடைய முடியும். இந்த திட்டப்பணியில் டாட்நெட்டின் மாயப்பொறியாகிய சி.எல்.ஆர்-யை புலம் நிரல்படுத்து வாயில் அணியில் நிறைவேற்றப்பட்டுள்ளது. புலம் நிரல்படுத்து வாயில் அணியின் பயன்பாடு அதன் நெகிழ்ச்சித் தண்மையும் மறு உருவாக்குத் தண்மையும் ஆகும். எனவே புலம் நிரல்படுத்து வாயில் அணி பயன்படுத்தப்படுகிறது. இவ்வாறு நிறைவேற்றப்படுவதன் மூலம் டாட்நெட்டை நோக்கி ஓடவேண்டிய நிரல்களின் ஓடும் நேரம் கணிசமாக குறைகிறது. இம்மேம்பாட்டை அடைய வன்பொருள் - மென்பொருள் இணை வடிவமைப்பு முறை கையாளப்பட்டுள்ளது.

LIST OF FIGURES		
FIGURE NO	TITLE	PAGE NO
3.1	Frequency of CIL instructions encountered in benchmark programs.	10
3.2	Flowchart for co-designed approach	11
3.3.	Overall Design of the System	12
3.4.	Flow Diagram for Method Invocation	17
3.5.	Hardware Design	18
4.1.	Entry Point Token Format	22
4.2.	Detailed Hardware Design	26
4.3.	State Diagram of the hardware system	28
5.1.	Snapshot depicting the initialization of the main method	34
5.2.	Snapshot depicting the call of method function (int, int)	35
5.3.	Snapshot depicting the return of the method and handling of branch condition	35
5.4	Snapshot depicting the execution of unconditional branch instruction	36
5.5.	Snapshot depicting the end of execution of the program	36
5.6.	Snapshot of the execution time of Pure Software Solution	37
5.7.	Snapshot of the exec time of Co-designed Solution	38
5.8.	Profiling results	39
A1.	XPS in action	44
A2.	Design Flow in XPS	45
A3.	XMD Command Shell	46
A4.	XMD Targets	46
A5.	CLI File Format	50

LIST OF ABBREVIATIONS

1. VM – Virtual Machines
2. FPGAs – Field Programmable Gate Arrays
3. CLR – Common Language Runtime
4. MSIL – Microsoft Intermediate Language
5. CIL – Common Intermediate Language
6. CLI – Common Language Infrastructure
7. ASICs – Application Specific Integrated Circuits
8. PLDs – Programmable Logic Devices
9. JVM – Java Virtual Machine
10. ISA – Instruction Set Architecture
11. AOT Compilation – Ahead of time Compilation
12. JIT Compilation – Just In Time Compilation
13. RVA – Relative Virtual Address
14. PE – Portable Executable
15. TOS – Top of Stack
16. TOS2 – Second element from the top of stack
17. FSL – Fast Simplex Link
18. JTAG – Joint Test Action Group
19. VHDL – Very high speed integrated circuit Hardware Description Language
20. ELF – Executable and Linking Format
21. FIFO – First In First Out
22. SCP – Soft-Core Processor

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION TO VIRTUAL MACHINES

The field of virtual machines (VM) and language-independent and platform-independent execution environments has always fascinated language designers and implementers for a long time. Such implementations have a lot of advantages over the native compilers strategy, the main objective of adopting such an approach being portability [1].

Portability is achieved by having the high level code translated to an intermediate form, which a system, usually called the Virtual Machine (VM), translates to the native code of the target architecture. The virtual machine is a software implementation that lies between the application and the operating system. Or in other words, the VM can be thought of as a program that can run other programs. The intermediate form acts as a means of communication between the high level front end and the low level back end. One of the ways of realizing such a virtual Machine system, and the widely adopted one, is by modeling the VM as a stack. The reason for adopting a stack-based architecture is that, any virtual machine, which aims to achieve portability across platforms, cannot make assumptions on the underlying architecture. A stack, on the other hand can offer higher level abstractions and abstract actions can be specified on a stack (push, pop, add-top-two etc.). Thus, it becomes desirable to model VMs as abstract stack machines. The next section gives an introduction to the .NET environment, which has a VM as one of its components.

1.2 INTRODUCTION TO .NET

The name “.NET” is misleading. The name likely comes from Microsoft’s original idea of web services. However, .NET is much more than that. It is a framework that includes an execution engine (i.e., a virtual machine) to allow applications to be platform-independent (like Java). Unlike Java, it is language independent. Programs can

be written in any .NET-supported language. The Common Language Environment (CLR) is the run-time environment of the .NET framework. It manages the execution of code and provides services that make the development process easier. The intermediate code form of the .NET system is called Common Intermediate Language (CIL) or the Microsoft Intermediate language (MSIL). A detailed review of the .NET architecture has been provided in Appendix A5.

1.3 .NET in EMBEDDED SYSTEMS

An exponential growth has been witnessed in the field of embedded systems, spawning the growth of embedded devices on a large scale. With the Internet boom, more and more embedded devices are being designed through which Internet access is possible. As this growth continues the devices are becoming more intelligent and complex than ever. So there is a significant increase in the amount of software that has to be written for such devices. Hence from a programmer's point of view it would be helpful if they can have a level of abstraction between the underlying hardware and the software, which is to be written so that they can concentrate more on the code rather than focusing on the low-level details of the embedded device. .NET provides this abstraction by helping a programmer concentrate on the implementation rather than focusing on the low level hardware details. Thus .NET technology may truly enable the embedded system revolution by providing a flexible and common platform.

But .NET in its present form cannot be directly used for programming embedded devices due to several reasons. One among them is that .NET programs generally run in an interpreted environment where the underlying Virtual Machine (VM), the Common Language Runtime (CLR) interprets each .NET instruction. This means that the VM translates the program into machine instructions that the processor in the device can execute. But such an interpreted execution takes time to complete. But time for execution is directly proportional to power consumption in an embedded system. So we cannot prefer such a purely interpreted embedded system since it consumes more power. So it is imperative for the programmer to write programs that are power efficient, i.e. they must consume less time to get executed. Another reason why .NET in the present form cannot

be used on embedded devices is that the size of the underlying VM (the CLR) is too large to be loaded into embedded devices, as memory is a constraint in these devices.

1.4 PROPOSED SCHEME

The very common solution to build any general VM is to have it as an interpreter and/or a JIT compiler or an AOT compiler. This kind of a software solution is relatively easier to build and cost effective, but it compromises on the performance. Moreover it also suffers from sequential execution. Pure hardware implementations are possible but the price they demand is flexibility. Moreover, complexity and cost are high for such an approach.

The objective is to exploit the advantages of both these approaches by employing both of them. In other words, a co-design sort of an approach to achieve a better negotiation between cost and performance is proposed. Also, with this paradigm lies flexibility too. Using FPGA provides a development environment for easily shifting the partitioning between hardware and software to arrive at an optimized solution in an iterative fashion. Hence a co-designed strategy is adopted to design the CLR.

To counter the huge size of CLR we have to consider a subset of the .NET ISA and to write a CLR which supports this subset. The subset has to be chosen in such a way that instructions in this subset help us to program the embedded devices without any constraints. Such a minimal .NET processor has been designed which can execute a subset of .NET ISA efficiently by exploiting the inherent features of FPGAs.

1.5 INTRODUCTION TO FPGAs

Field Programmable Gate Arrays (FPGAs) are an array of logic gates that can be hardware-programmed to fulfill user-specified tasks. In this way, one can devise special purpose functional units that may be very efficient for some limited task. It is also possible to customize the entire instruction processor at compile time or at run time. As FPGAs can be reconfigured dynamically, be it only 100 to 1,000 times per second, it is possible to optimize them for more complex special tasks at speeds that are higher than

what can be achieved with general purpose processors. An FPGA is similar to a Programmable Logic Device (PLD) but whereas PLDs are generally limited to hundreds of gates, FPGAs support thousands of gates. The underlying simplicity of FPGAs is that any circuit can be emulated with the available gates, and the user has to simply program the board by loading a bitstream, which instructs the board to configure itself.

FPGAs fill a gap between discrete logic and the smaller PLDs on the low end of the complexity scale and costly custom ASICs on the high end. Since they can be programmed, FPGAs can be used to design hardware systems by minimizing costs compared to ASICs as well as provide sufficient functionality to develop complex systems. They consist of an array of logic blocks that are configured using software. Programmable I/O blocks surround these logic blocks. Both are connected by programmable interconnects (Fig. 1). The programming technology in an FPGA determines the type of basic logic cell and the interconnect scheme. In turn, the logic cells and interconnection scheme determine the design of the input and output circuits as well as and memories are afforded by support for various single ended and differential I/O standards. Also found on today's FPGAs are system-building resources such as high speed serial I/Os, arithmetic modules, embedded processors, and large amounts of memory. Initially seen as a vehicle for rapid prototyping and emulation systems, FPGAs have spread into a host of applications. They were once too simple, and too costly, for anything but small-volume production. Now, with the advent of much larger devices and decline of per-part costs, FPGA's are off the prototyping bench and onto production.

1.6 EXISTING TECHNIQUES TO IMPROVE PERFORMANCE OF VIRTUAL MACHINES

Many advances have been made in order to have a faster execution environment. A few case studies are presented in this section that show how improvements of Java Virtual Machines (JVM) have been accomplished with the help of hardware and software. The differences between the JVM and CLR are also pointed out and it can be inferred that issues that are applicable for a hardware design of JVM cannot be applied for the hardware design of CLR.

1.6.1 METHODOLOGIES ADOPTED TO IMPROVE PERFORMANCE OF JVM

One way to achieve a faster runtime is by having dedicated Java Coprocessors, which can help the host processor execute Java code better, faster and in an efficient manner. Some examples are JVXtreme, JStar and Xpresso [2]. Embedded systems and real time systems, which execute Java Code, adopt this kind of a solution. Although this idea seems to be attractive, it is necessary to have a GPP running on the target machine. Moreover, this increases the cost too.

The other extreme is to have specific processors, which can directly execute Java Bytecodes. These processors are called Java Processors [3]. The beauty of these processors is that the very ISA of them is the JVM ISA, meaning that they work on bytecodes directly and run them directly on hardware, thereby achieving speedier execution. Moreover additional and time-consuming tasks of interpretation and JIT compilation are also done away with. Thus the performance benefits that can be reaped from such an approach are many compared to coprocessor technology.

Another way to speed Java execution on a standard RISC architecture is to extend the architecture itself, to directly execute Java instructions. ARM designers added a new Java instruction set to the classic ARM architecture [3]. The Java ISA is executed in a Java mode, which is entered on a branch. In the Java mode, the CPU executes Java byte code instructions. There have also been complete FPGA implementations of JVM. One such work is the implementation of JVM on FPGAs [4] that draws its inspiration from Sun's Pico Java Processor.

So, it is evident that the JVM has been given much attention by the embedded systems group and reconfigurable computing groups. .NET on the other hand, is an upcoming technology and not so much work has been done as regards its embedded development. Although a chip that is capable of running a very restricted subset of the CLR is already out in the markets, its scope of application is pretty narrow. So in this project a system that can have a broader application domain but with features that might typically be required by an embedded system has been built.

1.6.2 DIFFERENCES IN THE DESIGN ISSUES OF CLR AND JVM

Although the CLR and JVM are both stack-based architectures and share a few common features, there are a few vital differences between them. A few of those differences are presented here.

First of all, the JVM is so designed to run only Java code efficiently. Although it's quite possible to make the JVM coexist with other languages, the JVM is necessarily a sub-optimal multi-language platform [5, 6]; whereas, the .NET framework and the CLR has been designed from the ground up having language interoperability in mind. The assembly language of the CLR is an object-oriented assembly and includes generic instructions, but the JVM ISA has got no generic instructions. Another difference between the JVM and the CLR is that Java doesn't provide provisions for writing native code or type-unsafe features of typical programming languages (native pointers etc). But the .NET framework differentiates between managed and unmanaged code. The unmanaged code is capable of executing code, which can run out of the vigilance of the CLR. .NET adheres to the Virtual Object System (VOS). Value types can only be primitives in Java, where as the .NET supports structs and unions. CLR includes provisions for automatic boxing and unboxing, whereas JVM doesn't. The parameter passing conventions differ in Java and .NET.

Thus the various design issues and parameters are quite different for the JVM and the CLR. Those that apply for the JVM do not apply for the CLR, hence the need for a CLR specific design.

1.7 ORGANIZATION OF THE THESIS

The outline of this thesis is as follows. Chapter 2 discusses the works that have taken place in the field of reconfigurable computing and virtual machines. Chapter 3 describes the high-level and detailed design of the system. Chapter 4 explains the implementation details followed by the Chapter 5 that discusses the results obtained. Chapter 6 provides conclusion and the future work.

CHAPTER 2

LITERATURE SURVEY

This chapter describes the existing work that have taken place in the field of reconfigurable computing and virtual machines and the approaches that have so far been adopted to improve the performance of virtual machines.

2.1 JVM IMPLEMENTATION OF FPGAs

Nagendra Kumar *et al* have implemented Java Virtual Machine (JVM) on FPGAs [4]. In this work, a Java microprocessor core using FPGA technology is being experimented with and its preliminary functionality is tested and verified. The processor core eliminates the need for commonly used interpreters, JIT compilers and their overhead. The core accelerates the JVM runtime environment. It executes the most commonly used instructions in hardware. Complex instructions are micro coded, with the most complex ones trapped and emulated in software. This work also gives an idea of a “stack-cache” which is a new design methodology based on PicoJava Processor to accelerate the performance of the core.

2.2 THE PICOJAVA PROCESSOR

Olayinka Olabinjo *et al* offer insight into the hardware processor design of a stack-processor [2]. Applications written in Java are compiled to an intermediate representation before being sent to a client over the Internet or another network. Any processor and operating system combination that has an implementation of the Java Virtual Machine, either embedded in the operating system or in a browser, can execute these Java applications and produce correct results. This virtual machine comprises a specification of a file format for the executable (called a class file), an instruction set (Java byte codes), and other features such as threads and garbage collection. The main

factors that influence this design are portability, security, code size, and the ease of writing an interpreter or a JIT compiler for a target processor and operating system.

2.3 IMPLEMENTATION OF HARDWARE PARTITION ON A CO-DESIGNED JVM

Hejun Ma *et al* examine the various aspects of a development environment, the effects the environment's characteristics have on the hardware design and the factors that must be considered when making any design decisions [7]. The design of the hardware portion of the co-designed virtual machine is constrained by the target environment. The focus is to target desktop workstation that has an FPGA available through a local bus. Due to the target environment, there are several implications to the requirements of the hardware design, the primary one being the availability of resources. The other major design consideration is the Communication between the FPGA and the Host processor, which needs to be quick and efficient. The size and speed of the memory that is accessible also influences the design. All of the above mentioned factors contribute to the development environment and its suitability for a virtual machine.

It is clear that a development environment suitable for one virtual machine may not be suitable for another. The more instructions that can be implemented in hardware the better, since the overall purpose of the co-design is to obtain faster execution through pipelining the fetch-decode-execute loop.

2.4 REGISTER STACK ARCHITECTURE IN INTEL ITANIUM PROCESSORS

Scott Townsend *et al* discuss about the register stack architecture of the Itanium Intel Processors and its working, functionality and performance advantages [8]. A feature of processors in the Itanium processor family is the Register Stack Engine (RSE). The RSE is a hardware implementation inside the processor that helps a subset of the General Registers (GRs) implements the register stack by handling register overflows. Its main function is to act like the traditional memory stack, except much faster. Without the RSE implementation, code in a program calls a function, puts the parameters to pass to the function on the stack (in memory), and the receiving function must retrieve them from

the stack into registers to manipulate them. But in the Itanium processor family, the register stack enables extremely fast switching of the function call process with little to no overhead unlike Traditional Processors. Because of this there is a dramatic performance gain over the normal stack based machine.

2.5 PLATAFORMA .NET

The aim of this whitepaper is to analyze the portability of the .NET platform and the Internet revolution into the world of embedded systems [9]. The final goal would be to devise a hardware implementation of the .NET CLR so that CLI byte-code could be run natively. The particular objectives are

- Studying the JIT compilation for embedded processors, including high performance processors like ARM with extensive OS support, as well as processors used in rapid prototyping systems with small or non-existing OS support.
- Modifying the architecture of a given processor to improve execution of native code coming from JIT compilation of CLI.
- Studying the viability of a processor that partially (or completely) implements the .NET CLR. Given the complexity of the VOS object system and the .NET virtual machine, it is not intended, at first, to cover the entire CLR specification. Instead, a significant subset will be chosen, that could be used to run simple programs and interact with hardware devices in an embedded environment.

CHAPTER 3

DESIGN

This chapter discusses the design methodology adopted to construct the hardware and the software components of the project. First the overall methodology adopted is explained followed by the detailed design of the hardware and software components.

3.1 OVERALL DESIGN

3.1.1 THE CO-DESIGN METHODOLOGY

This methodology is adopted to exploit advantages of both Hardware and software implementations of VM. The first step is identifying those MSIL instructions that can be implemented on hardware. To identify which part of the CLR goes into the hardware and which part goes into the software a static profiling was done on benchmark programs like Splunc, FFT and Ray Tracer. Four class of instructions namely *arithmetic instructions(A)*, *loads for local variables(B)*, *loads for constants(C)* and *object handling instructions(D)* were identified and their frequency of occurrences were also found out.. The results show that all these four classes of instructions share the majority of the total instructions with almost equal percentage of occurrence and hence can be directly implemented in hardware.

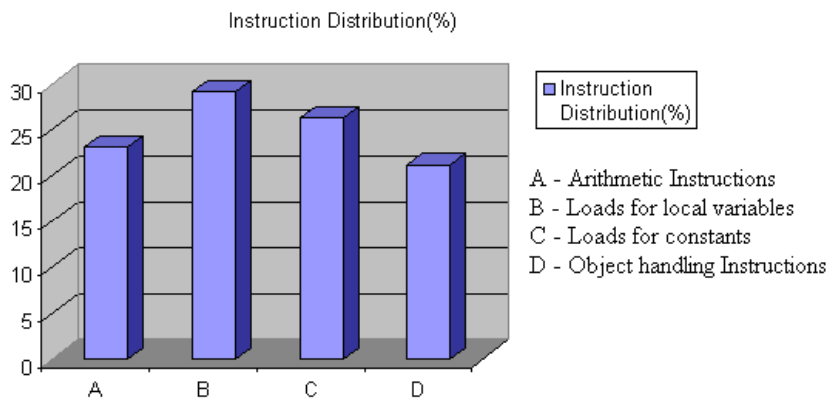


Figure 3.1: Frequency of CIL instructions encountered in benchmark programs.

The next step is to actually design the hardware for those identified instructions and porting them onto hardware, each instruction at a time. The system is then evaluated for performance and the above process is iterated by porting new instructions until the performance achieved is optimal. The flowchart (figure 3.2) depicts this process.

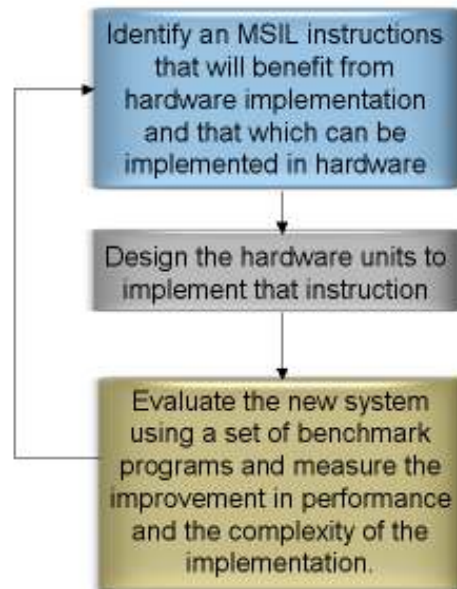


Figure 3.2: Flowchart for co-designed approach

The co-design methodology solves two purposes. Complex features of the CLR like object orientation, exception handling that are too complex to be run on hardware can be performed in software while simple accelerate-able code like loads and stores, arithmetic manipulations which lend themselves to parallelization can be implemented directly in hardware.

3.1.2 COMPONENTS OF THE SYSTEM

The following figure (Figure 3.3) below depicts the overall system. The basic components of the system are

- MicroBlaze Softcore Processor and main memory
- Custom Hardware encompassing the functional units and the 32 entry stack
- Processor and Custom Hardware Interface – The FSL Interface

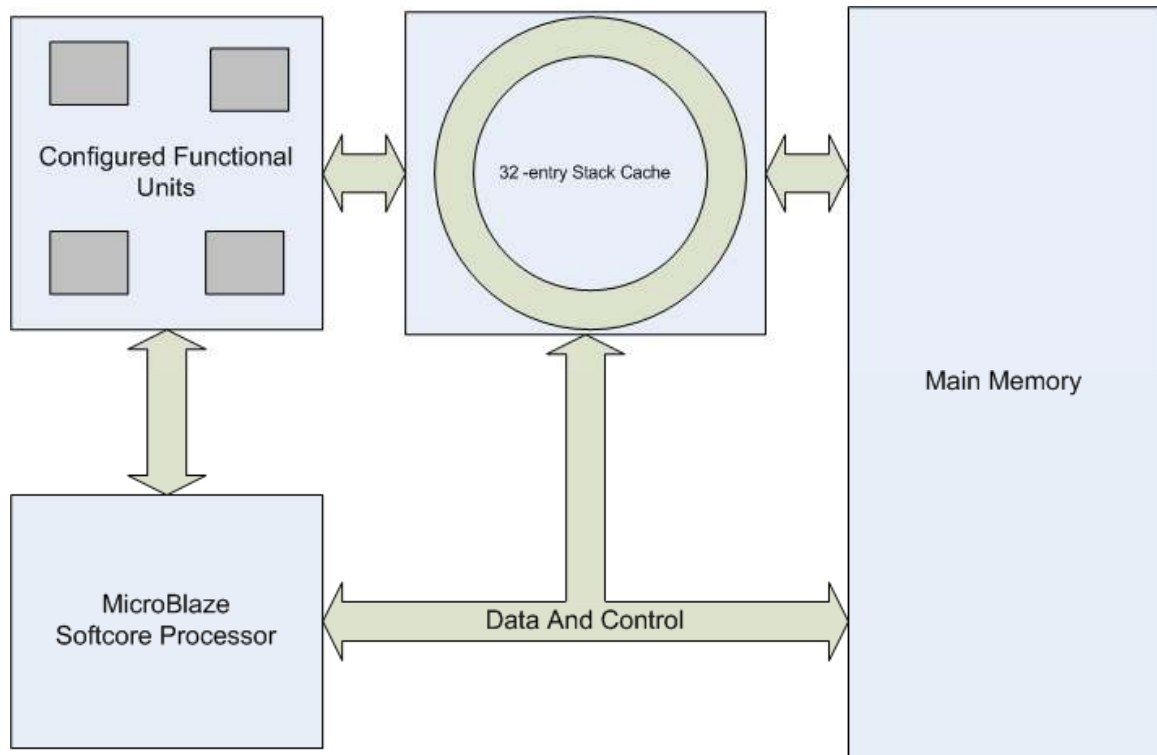


Figure 3.3: Overall Design of the System

3.1.2.1 MICROBLAZE SOFTCORE PROCESSOR

The MicroBlaze embedded soft core is a reduced instruction set computer (RISC) optimized for implementation on Xilinx field programmable gate arrays (FPGAs). The MicroBlaze embedded soft core is highly configurable, allowing users to select a specific set of features required by their design. This is a 300MHz 32-bit big-endian processor.

The processor's fixed feature set includes the following:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline

3.1.2.2 CUSTOM HARDWARE

Configured Logic Blocks (CLBs) contain flexible Look-Up Tables (LUTs) that implement logic plus storage elements used as flip-flops or latches. These CLBs are

utilized to design the custom hardware, which can either be a stack or an adder or a shift barrel multiplier to perform various required operations. The custom hardware primarily comprises of three components, namely

- A 32-entry Hardware Stack
- Control Unit
- Configured Functional Units

3.1.2.3 PROCESSOR AND CUSTOM HARDWARE INTERFACE

The Processor-Custom Hardware communication is achieved with the help of an existing Fast Simplex Link (FSL) interface which has been provided by Xilinx for communication with low latency between the hardware and the softcore processor. FSL is a unidirectional point-to-point communication channel bus used to perform fast communication between any two design-elements on the FPGA when implementing an interface to the FSL bus. Up to 8 master and slave FSL interfaces are available on the Xilinx MicroBlaze soft processor. These existing interfaces are used to transfer data to and from the register file on the processor to hardware running on the FPGA. Following are the features of FSL.

- Implements a unidirectional point to point FIFO-based communication
- Provide mechanism for unshared and non-arbitrated communication mechanism. This can be used for fast transfer of data words between master and slave implementing the FSL interface
- Provides an extra control bit for annotating data being transmitted. This control bit can be used by the slave- side interface for multiple purposes.
- FIFO depths can be as low as 1 and as high as 8K.
- Supports both synchronous and asynchronous FIFO modes. This allows the master and slave side of the FSL to clock at different rates.

FSL has I/O signals that must be activated by the hardware. These signals instruct how the data transfer happens between the softcore and the hardware.

The signals can be split into 3 categories. The first category of signals control the FSL bus while the next two categories refer to the signals that control the master and slave peripherals connected to the FSL bus.

3.1.2.3.1 SIGNALS THAT CONTROL THE FSL BUS

- **FSL_CLK**
This is the input clock to the FSL bus when used in the synchronous FIFO mode. The FSL_CLK is used as the clock for both the master and slave interfaces.
- **FSL_RST**
Output reset signal generated by the FSL reset logic. Any peripherals connected to the FSL bus may use this reset signal to operate the peripheral reset.
- **FSL_FULL**
Indicates FSL buffer is full
- **FSL_HASDATA**
Indicates FSL buffer has data

3.1.2.3.2 SIGNALS THAT CONTROL THE MASTER AND SLAVE PERIPHERALS CONNECTED TO THE FSL BUS

- **FSL_M_CLK/FSL_S_CLK**
This port provides the input clock to the master or slave interface of the FSL bus when used in the asynchronous FIFO mode.
- **FSL_M_DATA/FSL_S_DATA**
The data input to the master interface of the FSL bus and output to the slave interface of the FSL bus.
- **FSL_M_CONTROL/FSL_S_CONTROL**

Single bit control signal that is propagated along with the data at every clock edge by both the master and the slave.

- **FSL_M_WRITE**
Input signal that controls the write enable signal of the master interface of the FIFO. When set to '1', the values of FSL_M_Data and FSL_M_Control are pushed into the FIFO on a rising clock edge
- **FSL_S_READ**
Input signal on the slave interface that controls the read acknowledge signal of the FIFO. When set to '1', the values of FSL_S_Data and FSL_S_Control are popped from the FIFO on a rising clock edge
- **FSL_M_FULL**
Output signal on the master interface of the FIFO indicating that the FIFO is full.
- **FSL_S_EXISTS**
Output signal on the slave interface indicating that FIFO contains valid data

3.2 DETAILED DESIGN

The approach that has been adopted here is similar to a classical five-stage pipeline. The software part acts as a fetch and the decode unit, while the hardware part functions as an execute and write unit. The following sections explain the software, hardware and the interface design of the system.

3.2.1 SOFTWARE DESIGN

CLR engine, the software executor, forms the software partition and is responsible for the execution of CIL code. This engine is loaded inside the MicroBlaze softcore processor. After getting invoked, the CLR engine first loads the assembly (a standalone executable, similar to class file in Java) into the instruction buffer. This executable is referred to as a portable executable (PE). It's worth noting here that we are

running on a memory-restricted environment, in which we don't have the provision of loading very large assemblies. The memory needed, is necessarily a bottleneck. The CLR engine's core can be roughly portrayed by a large switch-case running in an infinite loop, as shown in the figure given below.

```
while (moreInstructions ())
{
    switch (next_instruction ())
    {
        case ADD:
            doAdd ();
            break;
        ...
        case PUSH:
            doPush ();
            break;
    }
}
```

The code written above is very similar to the software VM. More sophisticated techniques such as JIT or AOT are not employed here because a few of the instructions are to be implemented directly in hardware.

So when a .NET PE is given as the input to the CLR Engine, the program entry point is identified using the EntryPointToken present in the PE. Control information or the metadata information is embedded in the PE in the form of streams and tables. Each PE can have up to 43 tables and five streams. One such table is the MethodDef Table, which has information about the methods in the program. A row in the MethodDef table is indexed and the MethodRVA, which is the relative virtual address of the method to be executed is found. Next the method signature is identified. Once this is done we jump to the MethodRVA and read the Method Header. Method Header is classified into FAT Header and Tiny Header. Tiny Header is used incase the method has no local variables and arguments. In all other cases the FAT Header will be present. In the FAT header the LocalVarSigToken that will give us information about the number of local variables and type of the variables needed for this method is read. This process of initializing the metadata is explained in figure 3.4. The figure also explains how the flow of a method invocation takes place.

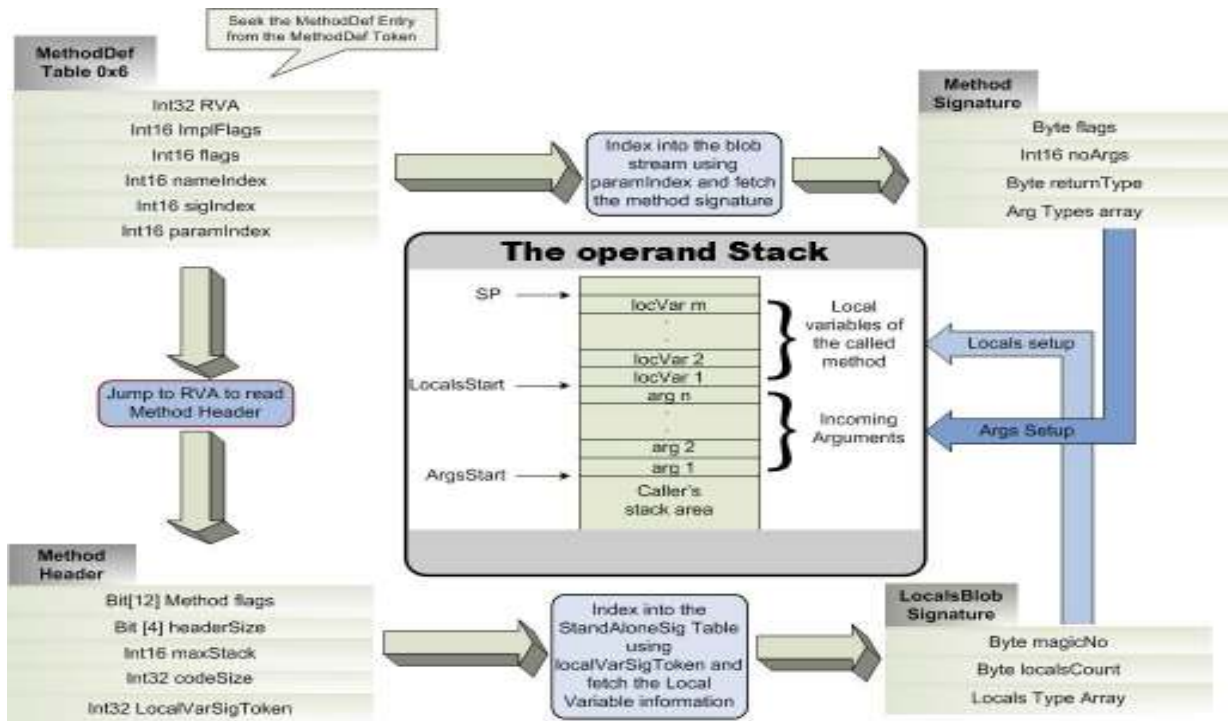


Figure 3.4: Flow Diagram for Method Invocation

The local variables and arguments of a method are initialized in the stack after which the CLR core gets executed. Each CIL instruction is read from the method body. Once the software engine decodes the instructions, the hardware system is alerted with the type of operation to be performed. This is performed by writing the interface with the opcode of the instruction read. The interface takes care of writing this value to the control unit in the hardware. The exception to this case are the load instructions where the data to be loaded is also written to the interface. The interface is configured to read and write both the control and data register in the hardware. Since the .NET is a stack-based machine, each CIL instruction manipulates the stack only. So the operands needed by an instruction are present either in the stack or in the data register of the hardware. Effectively the hardware becomes a self-contained entity with the software periodically alerting the hardware with the kind of operation that the hardware has to perform.

3.2.2 HARDWARE DESIGN

The hardware system consists of a stack, a control unit and functional units for performing operations. This section explains each of them in detail. These components are shown in figure 3.5.

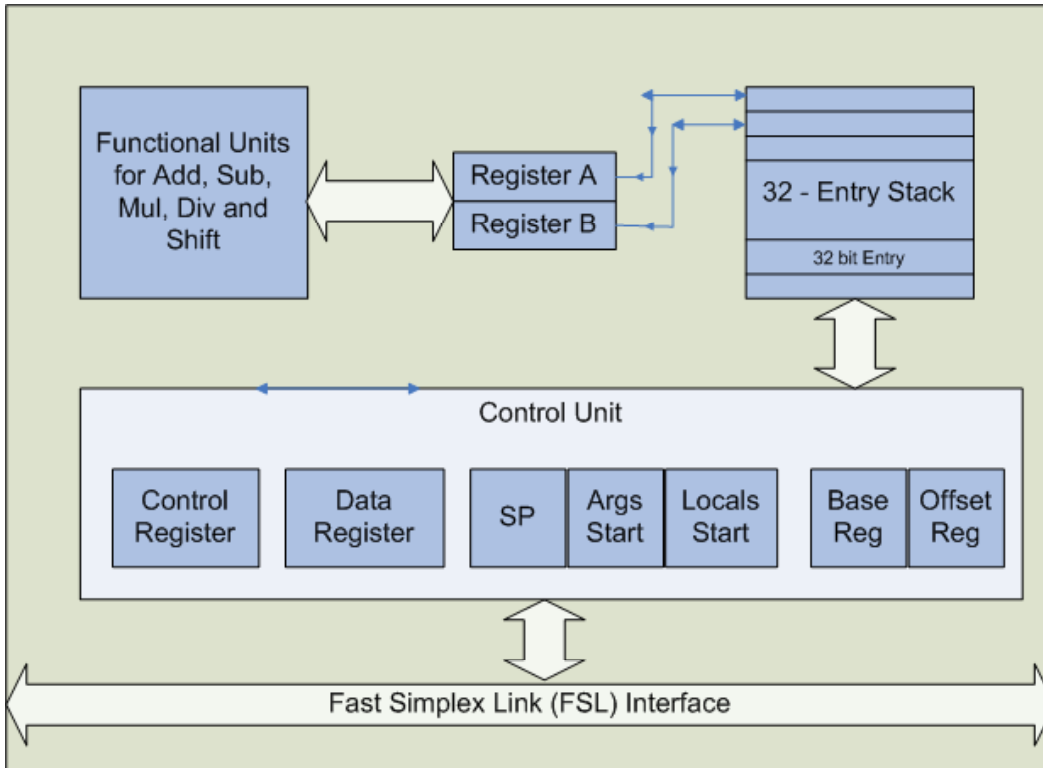


Figure 3.5: Hardware Design

3.2.2.1 CONTROL UNIT

The following are the components of a control unit.

- **Stack Pointer (SP)**
An index into the stack that identifies the top of the stack
- **ArgsStart - Argument Pointer**
A pointer to the stack that identifies the start of arguments passed to a method that is currently getting executed by the CLR engine.
- **LocalsStart – Locals Pointer**
A pointer to the stack that identifies the start of local variables passed to a method that is currently executed by the CLR engine.
- **Base Register**
A register that contains the base address of either the local variables or arguments passed to the function depending on the type of instruction currently getting executed.

- Offset Register

A register that contains an index into the stack from the address in the base register to access the corresponding local variable or the argument

- Control Register

This register contains the type of operation to be performed by the stack. This register is written by the softcore processor to instruct the stack to perform the required operation.

- Data Register

This register is written by the softcore processor. The value in this register is then written onto the stack by the hardware. The stack reads the control register and if it finds that the instruction is a load instruction then it reads the data to be loaded from the data register. Hence it is necessary for the software to write the data onto the data register by fetching it from the executable if the instruction it encounters is a load instruction. The exceptions are the load constants instructions which take the data based on an opcode directly and not from the data register.

The control register contains an opcode that identifies the type of operation to be done by the hardware whereas the data register has the data needed to fill the stack. When the CLR engine identifies the operation to be performed it writes the control register through the FSL link. In case of push instructions the CLR engine writes the data to be pushed onto the stack in the data register. The hardware identifies the type of operation to be performed and if it finds that a push needs to be done it uses the data present in the data register. Otherwise the hardware manipulates the elements of the stack. Base and offset registers have been provided for the purpose of random access on the stack.

3.2.2.2 HARDWARE STACK

The hardware stack is a 32-entry stack and each entry in the stack is a 32-bit register. The stack is modeled in lines of the Intel Itanium Processor [8]. It has been built in such a way that various operations that need to be done on the stack can be done by directly manipulating the stack entries, thereby preventing unnecessary pushes and pops.

To optimize stack accesses we have two registers (A and B) that contain the TOS and TOS2 of the stack (where TOS stands for Top of Stack). These two registers are used by the functional units to handle binary operations. These registers get loaded with the top of stack by default and when changes are made in the stack. This design helps us in saving time required to perform stack accesses.

Following are the operations that are implemented in Hardware

- *Push, Pop, Dup*
- *Load Constants (makes up app. 40 % of the instructions in an executable.)*
- *Load/Store args*
- *Load/Store locals*
- *Binary Operations: Add, Sub, BitAnd, BitOr, BitXor, BitNot, Negate*
- *Signed Compare Instructions*
- *Signed Branches and jumps*

Special mention is to be made on the branch instructions since they involve two-way data transfer from the stack to the software and vice versa. The result of the branch condition is intimated to the software, which uses this result to get the target instruction.

Load and store of local variables and arguments is done using the base and offset registers, which are pointers to the stack. For load instructions the stack[base + offset] is loaded onto the stack top and register A and B are appropriately changed while for store instructions the contents of register A is stores onto stack[base + offset]. While the offset register is initialized by the software, the base register is initialized with either the ArgsStart or the LocalsStart based on the type of opcode by the hardware (i.e.)

BaseRegister \leq LocalsStart if opcode is ldloc/stloc

BaseRegister \leq ArgsStart if opcode is ldarg/starg.

CHAPTER 4

IMPLEMENTATION

This chapter discusses the detailed implementation of the hardware and software components of the project. First the software implementation is explained in detail followed by hardware implementation.

4.1 SOFTWARE IMPLEMENTATION – CLR ENGINE

The CLR engine was implemented using C, which was compiled by the MicroBlaze C Compiler (mb-gcc). The main functions of this phase are bootstrapping of the .NET PE followed by metadata initialization. This is followed by the Core CLR loop which does the fetching and decoding of each instruction.

4.1.1 BOOTSTRAPPING

Initially, the .NET PE will be given as the input to the CLR engine. The binary dump of the executable is statically included in the file. This results in eliminating redundant disk accesses since the entire input is statically included in the input. All information about executable is included in the form of tables and streams. So the first step is to find out where the streams and tables are located in the executable. This is done by the bootstrapping phase. The steps in this phase are

1. Reading the DOS header to find if the given file is a valid WIN32 executable.
2. Once this validity is done the code and the data segment sizes are read.
3. Next step is to read the entry point followed by the base address of the code and the data section. These addresses are relative to the start of the executable and are called as the Relative Virtual Address (RVA).
4. After getting the RVAs of the code and the data section we read the section alignment values. Generally each section of the code is aligned to a 512-bit boundary.

5. Next step is to read the CLR header RVA and the CLR Header size.
6. Then we jump to CLR header RVA to read the metadata RVA and metadata size. Metadata RVA gives the information about where the tables and the streams are located.
7. Once the metadata RVA is read, we read the entry point token. The format of entry point token is shown below.

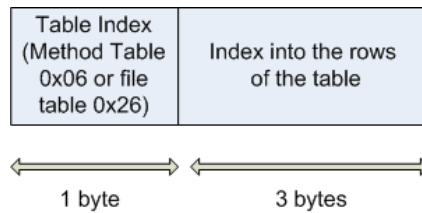


Figure 4.1: Entry Point Token Format

The entry point token serves in identifying the starting point of the execution. Generally the starting points of execution can either be a method or another assembly file. Current support has been offered for programs that have the starting point of execution in the current file. The entry point token is a 4-byte value that has the following information. The Most Significant Byte of the entry point token is a table index while the remaining 3 bytes index into the row of the table. The table that is indexed by the entry point token is the Method table that contains information about all the methods in the table. To identify which method is the starting point the remaining 3 bytes are used to index into the method table. Another important job of bootstrapping is to find the RVAs of all the streams that are present in the PE. So once the entry point token and the starting method to be executed are identified the metadata is initialized.

4.1.2 METADATA INITIALIZATION

As mentioned earlier the metadata information is organized in the form of tables and streams. The following are the data structures used to initialize the metadata.


```

Metadata
{
    Int64 tables_bit_vector;//A bit vector indicating
                                which tables are present
    Int32 no_of_tables;
    Int32 sorted_tables_bit_vector;
    Int32 no_of_rows[64]);//Rows in each table
}metadata;

```

This data structure stores the no of tables that are present in the PE. A maximum of 43 tables can be present in the PE. A table bit vector that contains which of these 43 tables is present is also stored. This data structure also contains information regarding the number of rows present in each of these 43 tables.

```

MethodDefEntry
{
    Int32 RVA;//RVA of the current method
    Int16 implFlags;
    Int16 flags;
    Int16 nameIndex;//Index into the string stream
    Int16 sigIndex;//Index into the blob stream
    Int16 paramIndex;//Index into the parameter
                                table.
}methodDefEntry;

```

This data structure is used to store information about each of the rows in the method table. The RVA data field in this data structure stores the RVA of each of the method present in the executable. The nameIndex field is an index into the “string” stream that gives the name of the method corresponding to this row in the method table. The paramIndex field is an index into the parameter table while the sigIndex is an index into the blob heap, which gives information regarding the signature of this method.

```

MethodFatHeader
{
    Int16 flagsAndSize;//Specify the header type (FAT
                                or TINY)
    Int16 maxStack;//No of stack entries this method
                                uses
    Int32 codeSize;//(in bytes)
    Int32 localVarSigToken;//Index into StandAloneSig
                                table to get local variable information
}methodFatHeader;

```

Each method has a method header that gives information about the method to be executed. Method headers can be classified into Fat and Tiny headers. Tiny headers are for those methods that have no arguments and have no stack requirements while the fat header is used for ordinary methods. This information is present in the first two bytes of

the header followed by the maximum stack that this method will consume. Following maxStack is the codesize information. Next is the local variable token that contains information like the number of local variables present and the type each local variable present.

```
LocalsBlobSignature
{
    Byte size; //The no of bytes occupied by this
               stream for the current method
    Byte magicNumber; //0x07
    Byte localsCount; //The number of local variables.
    // followed by the variables themselves
}localsBlobSig;
```

This data structure gives information regarding the local variables for each of the methods present in the method table. The first variable named size contains the size of this structure followed by the magic number (0x07). Next is the no of local variables present in this method followed by the type of each of the local variables present in the method

```
MethodSignature
{
    Byte byteCount; //The no of bytes occupied by
                   this stream for the current method
    Byte flags;
    Int32 noArgs; //The number of arguments for this
                 method
    Byte returnType;
    Int32 argsStartRVA; //The RVA of the argument list
}methodSig;
```

This data structure gives information regarding the signature of each method getting executed. The first information is the size of this structure followed by flags. Next information that is present is the number of arguments of each method and the return type of each method. Next we have argsStartRVA that is an index that contains the type of each argument passed to the function.

After initializing the metadata we move onto the core CLR loop.

4.1.3 CORE CLR LOOP

The function of this module is to fetch the instruction to be executed next, decode it and interrupt the hardware asking it to execute the instruction. But before this process

the stack must be initialized with the arguments and local variables for this method. A point that needs to be mentioned here is that we implement the operand stack in hardware while the control stack is taken care of by the software. The sequence of operations performed here are

1. The argsStart and localsStart registers are initialized
2. The stack is initialized with arguments to the method followed by the local variables themselves.
3. The next instruction from the input stream is read and decoded.
4. If the instruction has an immediate operand it is read and both the opcode and the operand are written to the control and data register of the hardware.
5. If a function call occurs, the values of SP, argsStart and localsStart are stored in the control stack of the software. The function token, the headers, the data structures are read and switching is done to Step1
6. If a return is encountered, the locals and arguments of the stack are flushed, the modified SP is loaded back, the values of argsStart and localsStart of the callee function are copied back onto the stack.
7. When the function ends, checking is done to ensure that the stack is in the same state as it was before the function call.

When the program completes execution the values of SP and the control registers are checked to be -1 and zero respectively. This is confirmed by printing the stack trace of the output for each instruction and at the end we can confirm that SP has been set back to -1.

4.2 HARDWARE IMPLEMENTATION

The hardware implementation consists of the hardware stack, the functional units and the control unit. The implementation was done in Very High Speed Integrated Circuit Hardware Description Language (VHDL). This file is then compiled and combined along with the software module, which is in the form of an Executable and Linking Format (ELF), and a bitstream is generated. This bitstream is downloaded onto the board and it is run. The hardware implementation consists of the following blocks.

1. Control Unit
2. Stack Unit
3. Functional Units

The following figure 4.2 explains the hardware system in detail

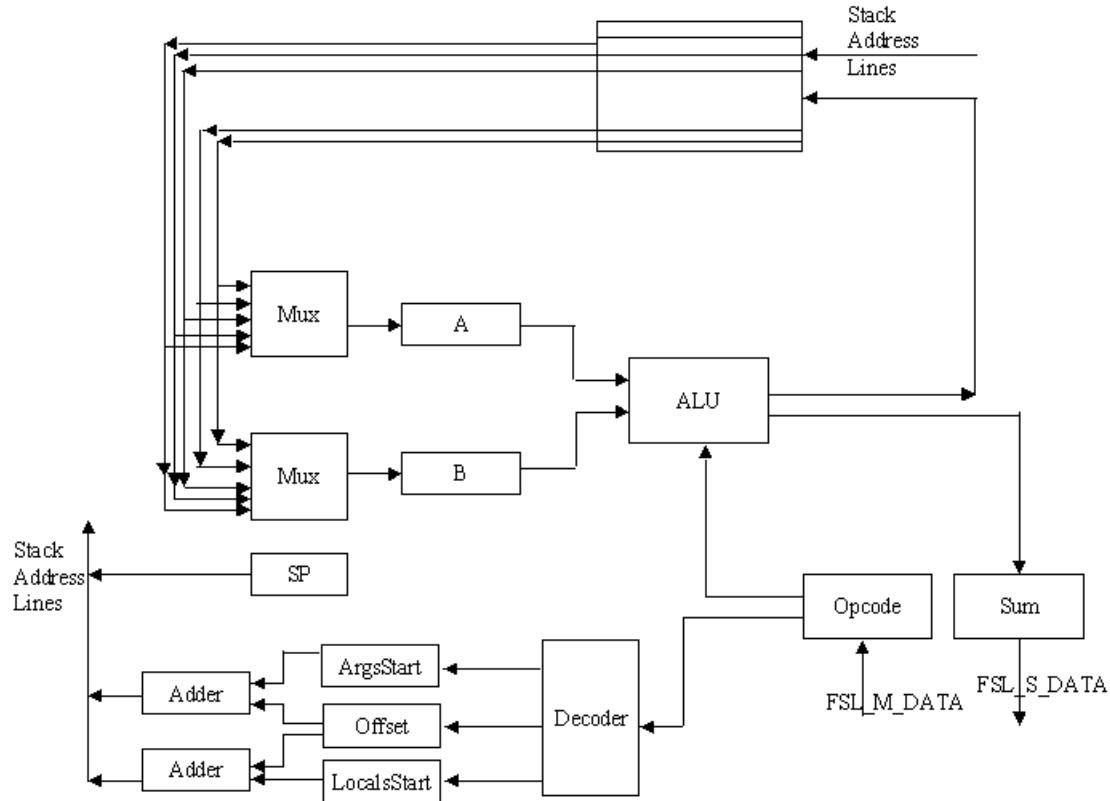


Figure 4.2 Detailed Hardware Design

4.2.1 CONTROL UNIT

Control unit consists of the Stack Pointer (SP), ArgsStart Pointer, LocalsStart Pointer, the base and offset registers and the control and the data register. Each of these is a 5-bit register. All these registers are written by the FSL interface and read by the stack. The initialization of these registers is by writing the control register with the opcode GetLocalArgs meaning that the argsStart and the localsStart are initialized with the values in the data register in a single clock cycle. This is achieved by encoding the data register with both these values and while fetching from the data register we decode it as follows.

```
when GetLocsArgs=>
    localsStart <= op(27 to 31);
```

```

argsStart <= op(19 to 23);
state <= idle;

```

When the control register is written by the CLR engine, the control unit decodes the value. If the opcode written refers local variables or arguments then the base and offset registers are used to perform random access on the stack. If the opcode received is ldloc/stloc then the base register is initialized with localsStart. Else if the opcode received is ldarg/starg then the base register is initialized with argsStart. The following code snippet explains this.

```

if (unsigned(FSL_S_Data) >= 37 and unsigned(FSL_S_Data) <=
40) then -- ldargs's
    base <= argsStart;
    offset <= FSL_S_Data(27 to 31) - 37;
    state <= Load_From_Stack1;
elseif (unsigned(FSL_S_Data) = 41) then -- ldarg.s
    base <= argsStart;
    state <= Read_Operands;
elseif (unsigned(FSL_S_Data) >= 42 and unsigned(FSL_S_Data)
<= 45 ) then -- stloc's
    sp <= sp - 1;
    base <= localsStart;
    offset <= FSL_S_Data(27 to 31) - 42;
    state <= Store_Into_Stack;
elseif (unsigned(FSL_S_Data) = 46) then-- stloc.s
    base <= localsStart;
    state <= Read_Operands;

```

4.2.2 STACK UNIT

The hardware stack has been implemented as an array of 32 registers, each being a 32-bit register.

```

constant MAXSTACK:INTEGER := 32; -- the max stack length
type stack_file is array (MAXSTACK-1 downto 0) of
std_logic_vector (0 to 4);
signal stack :stack_file;

```

This declaration creates a 32 entry 32-bit stack. The hardware also includes two temporary registers (A and B), which act as source registers for the functional units. These registers are initialized with the top two data from the stack in each clock cycle. So when an arithmetic operation is to be performed the top two elements would have been already available as source registers for the functional unit. This implementation helps in saving two pops that must occur for each operation to be performed. Since these registers are updated in each clock cycle any change in the top two elements in the stack will be

automatically reflected in each of the two registers. The following state machine describes the operation of the stack in detail.

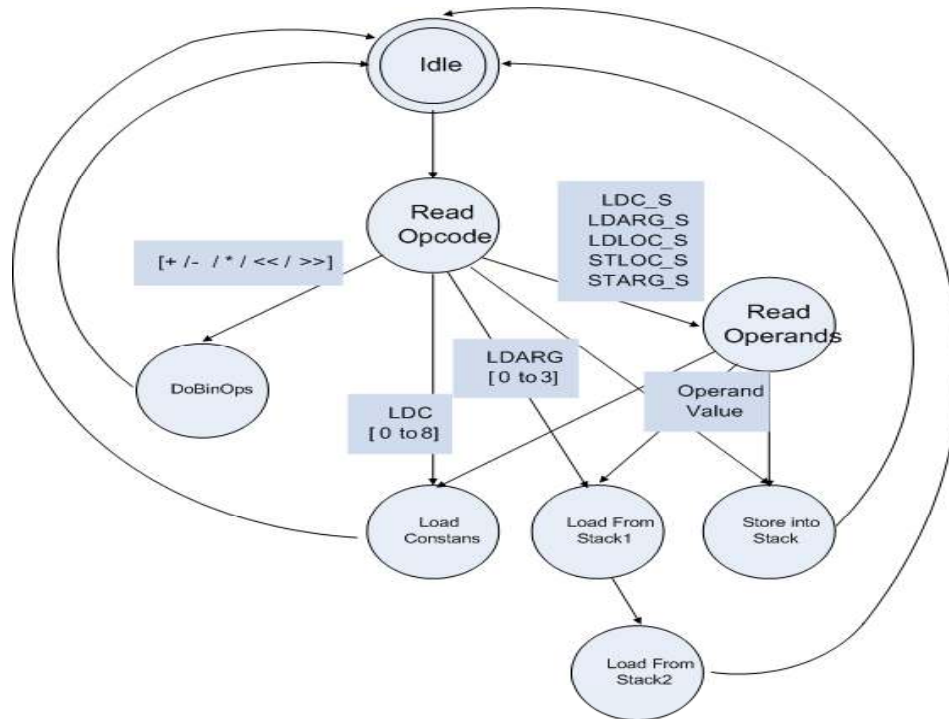


Figure 4.3: State Diagram of the hardware system.

The hardware is initially in the “idle” state. On getting the opcode it moves to the read opcode state. Based on the opcode read it moves to either the DoBinOps state or Load Constants state or the Read Operands state or Load from Stack1 state. In the read operand state it reads the immediate value of an instruction that is fetched by the CLR and written in the data register. From the read operands stage it moves to either of load constant or load from stack or store from stack state. There is a temporary stage transfer from the load from stack1 state to the load from stack2 state before we move back to the idle state.

4.2.3 FUNCTIONAL UNITS

The functional units perform basic arithmetic operations and bitwise operations on the data. The following are the functional units that have been designed.

1. Additions and Subtractions
2. Multiplication

3. All Branch instructions
4. Compare Instructions
5. Bit Manipulation instructions like (AND, OR, XOR, NEGATE, NOT).

All these units take the values of registers A and B as source operand and put the result back into the stack except for branches that give the result back to the CLR engine. The result for branch instruction specifies the outcome of the branch comparison, which is used by the CLR engine to calculate the branch target instruction.

4.3 INTERFACE ACCESS

Access to the FSL buffer is via the `microblaze_bread_datafsl()` and `microblaze_bwrite_datafsl()` calls. Here `bread` and `bwrite` refers to blocking read and write meaning that the slave waits till the data is available in the FSL buffer and the master waits till the FSL buffer becomes free. The parameters to this function are the FSL interface number and the data to be written in the buffer.

4.3.1 FSL WRITE OPERATION

When the data in `FSL_M_Data` and control bit in `FSL_M_Control` are ready to be pushed into the FIFO, the `FSL_M_Write` signal is set to '1' for one clock cycle. This will push the Data and Control signals onto the FIFO. If the FIFO is not implemented with BRAMs the data becomes available to the slave FSL interface as `FSL_S_Data` and the control becomes available as `FSL_S_Control` after the write clock edge. Further, the `FSL_S_Exists` signal is set to '1' to indicate data exists in the FIFO.

The following code snippet describes the sequence of operations to be performed when the push opcode is encountered.

```
int FSL_Push(int toPush)
{
    int j = 0;
    microblaze_bwrite_datafsl(2,0); //the opcode is
        written in the control register
    microblaze_bwrite_datafsl(toPush,0); //the data to be
        loaded is written in data register.
    microblaze_bread_datafsl(j,1);
    sp ++; //increment stack pointer
}
```

```
        return j;  
    }
```

4.3.2 FSL READ OPERATION

The read side of the FSL bus is controlled by the FSL_S_Read signal. When data is available in the FSL bus (FSL_S_Exists = '1'), the data in FSL_S_Data and the control bit in FSL_S_Control are immediately available to be read by the slave on the FSL bus. Once the slave completes the read operation, the FSL_S_Read signal has to be set to '1' for one clock cycle acknowledging that a Read has successfully been completed by the slave. After the clock edge where the read takes place, the FSL_S_Data and FSL_S_Control are updated with new data and FSL_S_Exists and FSL_M_Full are updated. The next chapter gives the results and performance comparison of the co-designed strategy and a pure software strategy.

CHAPTER 5

RESULTS

5.1 THE SYSTEM

The system used for testing is a Xilinx Virtex II Pro FPGA Board. This board has a 300MHz softcore MicroBlaze processor that employs 32-bit big-endian format. The board has a 32KB on-chip memory and a 256MB off-chip DDR memory. A .NET PE is given as an input to the CLR engine. After identifying the initial entry point, the CLR engine works asynchronously with the hardware to execute the PE. Since .NET Virtual Machine (VM) is a stack-based machine and each instruction is manipulated on stack alone, the design prints the stack trace of the executable as each instruction is executed. The Stack trace will contain the current value associated to the Stack Pointer (SP) along with the instructions it points to. The trace also depicts details such as the Relative Virtual Address (RVA) of the function invoked, its size, number of arguments in the method and the starting address of these arguments in the stack. The output can be seen in the HyperTerminal. Stack-trace as the output is a proof of the fact that the PE got executed.

5.2 HYPER TERMINAL SETTINGS

The Board is connected through RS232 cable and a JTAG cable. A Hyper Terminal was opened with the following configuration.

```
Connect Using: COM1 port  
Bits per second: 9600  
Data bits: 8  
Parity: None  
Stop bits: 1  
Flow control: Hardware
```

5.3 TEST CASES

The input to the CLR engine is the executable of the C# code. The input executable is disassembled to Microsoft Intermediate Language (MSIL) and shown below.

```
.method private hidebysig static void Main() cil managed
```

```

{
    // Code size: 41 bytes
    .maxstack 2
    .locals ([0] int32 V_0,[1] int32 V_1)
    IL_0000: ldc.i4.s    0x0a//10
    IL_0002: stloc.0
    IL_0003: ldc.i4.s    0x14//20
    IL_0005: stloc.1
    IL_0006: ldloc.0
    IL_0007: ldloc.1
    IL_0008: add
    IL_0009: stloc.0
    IL_000a: ldloc.0
    IL_000b: ldloc.1
    IL_000c: call int32
    InsideCSharp.HelloWorldConsoleApp::function(int32,
int32)

    IL_0011: stloc.0
    IL_0012: ldloc.0
    IL_0013: ldloc.1
    IL_0014: bge.s      IL_001c
    IL_0016: ldloc.0
    IL_0017: ldloc.1
    IL_0018: sub
    IL_0019: stloc.0
    IL_001a: br.s      IL_0024
    IL_001c: ldloc.0
    IL_001d: ldloc.1
    IL_001e: blt.s     IL_0024
    IL_0020: ldloc.0
    IL_0021: ldc.i4.1
    IL_0022: sub
    IL_0023: stloc.0
    IL_0024: ldloc.0
    IL_0025: ldc.i4.1
    IL_0026: add
    IL_0027: stloc.0
    IL_0028: ret
} //end of method InsideCSharp.HelloWorldConsoleApp::Main

.method public hidebysig static int32
function(int32 a, int32 b) cil managed
{
    // Code size      8 (0x8)
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: ldarg.0
    IL_0001: ldarg.1
    IL_0002: sub
    IL_0003: stloc.0
    IL_0004: br.s      IL_0006

    IL_0006: ldloc.0
    IL_0007: ret
} // end of method HelloWorldConsoleApp::function

```

The C# equivalent to above MSIL code is

```
namespace InsideCSharp {
    class HelloWorldConsoleApp {
        static void Main() {
            int i=10,j=20;
            i=i+j;
            i = function(i, j);
            if(i<j)
                i=i-j;
            else if(i>=j)
                i=i-1;
            i++;
        }
        public static int function(int a, int b) {
            return a-b;
        }
    }
}
```

The input case above encompasses all types of instructions that are being implemented in hardware such as Load constants, Add, Multiply, Subtract, Conditional and Unconditional Branching, and Functions handling.

Following is the Stack Trace of the above code executed on the Xilinx Virtex II pro board **Figure 5.1** provides the stack trace of the function main (). The initialization of Relative Virtual Address (RVA), Number of arguments to the method, Starting address of the arguments and size of the function are depicted in figure 1.

```
Output - HyperTerminal
File Edit View Call Transfer Help
No. of arguments for this method :: 00000000
argsStart for this method    :: 00000001

RVA:: 00000250
MaxStack                      :: 00000002
CodeSize                      :: 00000029
LocalVarSigToken              :: 11000001
Cur sp: 2
Cur sp: 2
Pushing 10 onto stack
Cur sp: 3
Stloc.0 done
Cur sp: 2
Pushing 20 onto stack
Cur sp: 3
Stloc.1 done
Cur sp: 2
ldloc.0 done
Cur sp: 3
ldloc.1 done
Cur sp: 4
Add done
Cur sp: 3
Stloc.0 done
Cur sp: 2
ldloc.0 done
```

Figure 5.1: Snapshot depicting the initialization of the main method.

Figure 5.2 depicts the trace of call to another function named “function” and the initialization of variables and operations on them for that function. Figure 5.1 is shown below

```

Output - HyperTerminal
File Edit View Call Transfer Help
Cur sp: 3
ldloc.1 done
Cur sp: 4
Add done
Cur sp: 3
Stloc.0 done
Cur sp: 2
ldloc.0 done
Cur sp: 3
ldloc.1 done
Cur sp: 4
CALL 06000002
No. of args for this method :: 00000002
argsStart for this method  :: 00000003

RVA:: 00000288
MaxStack                :: 00000002
CodeSize                 :: 00000008
LocalVarSigToken        :: 11000002
Cur sp: 5
Cur sp: 5
ldarg.0 done
Cur sp: 6
ldarg.1 done
Cur sp: 7
Sub done
Connected 0:33:07 Auto detect 9600 B-W-1 SERIAL COM1 N/A Capture Print echo

```

Figure 5. 2: Snapshot depicting the call of method function(int,int)

Figure 5.3 depicts the trace of the control getting back to the main function and then a Branching statement “greater than or equal to” getting executed.

```

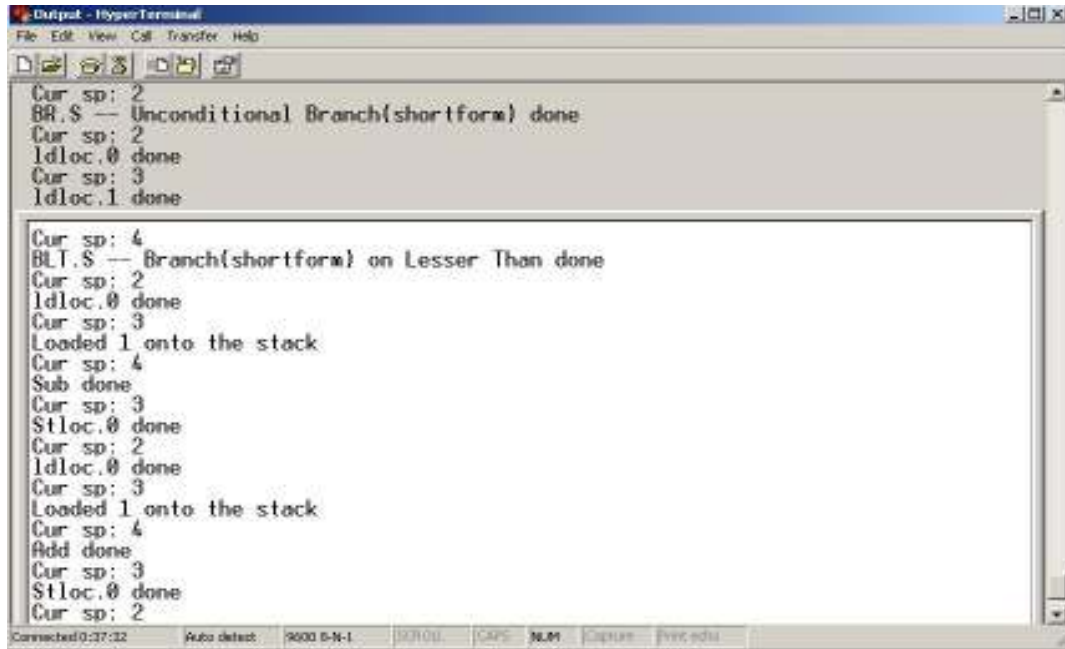
Output - HyperTerminal
File Edit View Call Transfer Help
Stloc.0 done
Cur sp: 5
BR.S -- Unconditional Branch(shortform) done
Cur sp: 5
ldloc.0 done
Cur sp: 6

Return @ i = 0008 = codeSize = 0008 : (A:00000003, L:00000005)
Cur sp: 3
Called function: A void return..
Cur sp: 3
Stloc.0 done
Cur sp: 2
ldloc.0 done
Cur sp: 3
ldloc.1 done
Cur sp: 4
BGE.S -- Branch(shortform) on Greater Than and Equal to done
Cur sp: 2
ldloc.0 done
Cur sp: 3
ldloc.1 done
Cur sp: 4
Sub done
Cur sp: 3
Stloc.0 done
Connected 0:36:18 Auto detect 9600 B-W-1 SERIAL COM1 N/A Capture Print echo

```

Figure 5.3: Snapshot depicting the return of the method function() and handling of branch condition

Figure 5.4 depicts the execution of an unconditional branch as well as a subtraction and addition operation

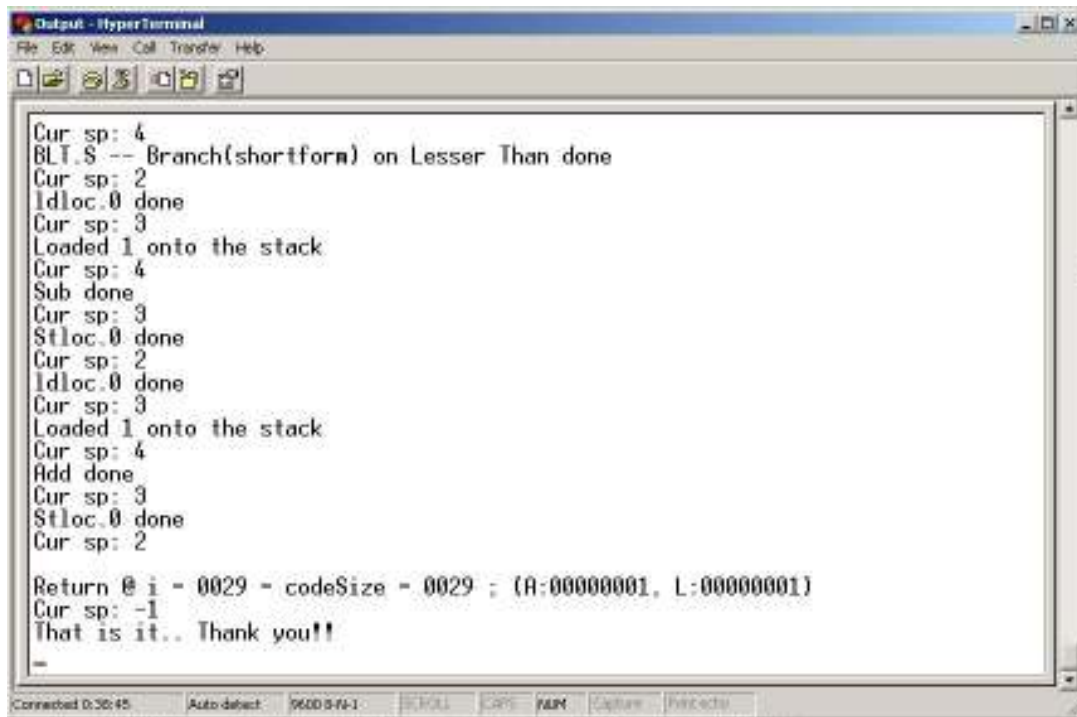


```
Output - HyperTerminal
File Edit View Call Transfer Help
Cur sp: 2
BR.S -- Unconditional Branch(shortform) done
Cur sp: 2
ldloc.0 done
Cur sp: 3
ldloc.1 done

Cur sp: 4
BLT.S -- Branch(shortform) on Lesser Than done
Cur sp: 2
ldloc.0 done
Cur sp: 3
Loaded 1 onto the stack
Cur sp: 4
Sub done
Cur sp: 3
stloc.0 done
Cur sp: 2
ldloc.0 done
Cur sp: 3
Loaded 1 onto the stack
Cur sp: 4
Add done
Cur sp: 3
stloc.0 done
Cur sp: 2
Connected 0:27:32 Auto detect: 9600 8-N-1 B310L CAPS NUM Capture Print
```

Figure 5.4 Snapshot depicting the execution of unconditional branch instruction

Figure 5.5 depicts the return of the main method and the end of execution of the program with the stack being reset to the original condition.



```
Output - HyperTerminal
File Edit View Call Transfer Help
Cur sp: 4
BLT.S -- Branch(shortform) on Lesser Than done
Cur sp: 2
ldloc.0 done
Cur sp: 3
Loaded 1 onto the stack
Cur sp: 4
Sub done
Cur sp: 3
stloc.0 done
Cur sp: 2
ldloc.0 done
Cur sp: 3
Loaded 1 onto the stack
Cur sp: 4
Add done
Cur sp: 3
stloc.0 done
Cur sp: 2

Return @ i = 0029 - codeSize = 0029 ; (R:00000001, L:00000001)
Cur sp: -1
That is it.. Thank you!!

Connected 0:30:45 Auto detect: 9600 8-N-1 B310L CAPS NUM Capture Print
```

Figure 5.5: Snapshot depicting the end of execution of the program.

5.4 PROFILING

The profiled output after porting the Software interpreter on Hardware without Custom Hardware Support is shown in figure 5.6 and the profiled output after porting the CLR on Hardware with Custom Hardware support is shown in figure 5.7.

The function main2() does the core job of execution. A comparison of the time taken by this function with and without Hardware support is done and the overall Speed Up is calculated. The values for the pure software and co-designed solution in microseconds are 10467.20 and 4757.72 respectively per function call.

Flat profile:

Each sample counts as 5e-05 seconds.

%	cumulative	self	self	total	name
time	seconds	seconds	calls	us/call	us/call
77.94	4.68	4.68			XUartLite_SendByte
8.36	5.18	0.50			__modsi3
4.21	5.44	0.25			_mbtowc_r
2.78	5.60	0.17	1	167228.34	222208.72 main
1.38	5.69	0.08			_vfprintf_r
1.08	5.75	0.07			__sfvwrite
0.54	5.78	0.03			memmove
0.36	5.81	0.02			__umoddi3
0.36	5.83	0.02			outbyte
0.35	5.85	0.02			memchr
0.29	5.87	0.02			fflush
0.28	5.88	0.02			__udivdi3
0.23	5.90	0.01	1	13573.24	49473.68 executeMethod
0.23	5.91	0.01			__umodsi3
0.21	5.92	0.01	427	29.45	54.12 my_f_read
0.20	5.93	0.01			__udivsi3
0.19	5.95	0.01			write
0.18	5.96	0.01	690	16.01	16.01 readByteFromStream
0.15	5.97	0.01	248	36.59	36.59 printStackTrace
0.13	5.97	0.01			__mulsi3
0.12	5.98	0.01			__swrite
0.10	5.99	0.01			puts
0.06	5.99	0.00			strlen
0.04	5.99	0.00			XUartLite_RecvByte
0.04	5.99	0.00	145	16.72	16.72 pop
0.03	6.00	0.00	75	21.33	21.33 my_f_seek
0.02	6.00	0.00			fprintf
0.02	6.00	0.00	1	1349.83	2073.68 updateMetadata
0.02	6.00	0.00	172	6.47	6.47 push
0.02	6.00	0.00			printf
0.01	6.00	0.00	32	20.70	36.71 my_f_getc
0.01	6.00	0.00	12	54.16	54.16 getOffsetForTable
0.01	6.00	0.00			my_f_eof
0.00	6.00	0.00			getTos2
0.00	6.00	0.00	24	10.42	10.42 my_f_tell
0.00	6.00	0.00	6	41.66	258.14 getMethodFatHeaderFromStream
0.00	6.00	0.00			__fixdfsi
0.00	6.00	0.00			__muldi3
0.00	6.00	0.00	6	25.00	187.36 getLocalsBlobSignatureFromStream
0.00	6.00	0.00			_malloc_r
0.00	6.00	0.00	1	112.49	112.49 initStack
0.00	6.00	0.00			__divsi3
0.00	6.00	0.00			puts r
0.00	6.00	0.00	1	99.99	10467.20 main2
0.00	6.01	0.00			_exception_handler

0.00	6.01	0.00				_printf_r
0.00	6.01	0.00	6	8.33	333.06	getMethodDefEntryFromStream
0.00	6.01	0.00				_sinit
0.00	6.01	0.00				localeconv
0.00	6.01	0.00				__smakebuf
0.00	6.01	0.00				__calloc_r
0.00	6.01	0.00				setlocale
0.00	6.01	0.00	6	0.00	248.23	getMethodSignature
0.00	6.01	0.00	1	0.00	49994.71	execute

Figure 5.6: Snapshot of the execution time of Pure Software Solution

Flat profile:

Each sample counts as 5e-05 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
77.94	4.68	4.68				XUartLite_SendByte
8.36	5.18	0.50				__modsi3
4.21	5.44	0.25				__mbtowc_r
2.78	5.60	0.17	1	167228.34	222208.72	main
1.38	5.69	0.08				__vfprintf_r
1.08	5.75	0.07				__sfvwrite
0.54	5.78	0.03				memmove
0.36	5.81	0.02				__umoddi3
0.36	5.83	0.02				outbyte
0.35	5.85	0.02				memchr
0.29	5.87	0.02				fflush
0.28	5.88	0.02				__udivdi3
0.23	5.90	0.01	1	13573.24	49473.68	executeMethod
0.23	5.91	0.01				__umodsi3
0.21	5.92	0.01	427	29.45	54.12	my_f_read
0.20	5.93	0.01				__udivsi3
0.19	5.95	0.01				write
0.18	5.96	0.01	690	16.01	16.01	readByteFromStream
0.15	5.97	0.01	248	36.59	36.59	printStackTrace
0.13	5.97	0.01				__mulsi3
0.12	5.98	0.01				__swrite
0.10	5.99	0.01				puts
0.06	5.99	0.00				strlen
0.04	5.99	0.00				XUartLite_RecvByte
0.04	5.99	0.00	145	16.72	16.72	pop
0.03	6.00	0.00	75	21.33	21.33	my_f_seek
0.02	6.00	0.00				fprintf
0.02	6.00	0.00	1	1349.83	2073.68	updateMetadata
0.02	6.00	0.00	172	6.47	6.47	push
0.02	6.00	0.00				printf
0.01	6.00	0.00	32	20.70	36.71	my_f_getc
0.01	6.00	0.00	12	54.16	54.16	getOffsetForTable
0.01	6.00	0.00				my_f_eof
0.00	6.00	0.00				getTos2
0.00	6.00	0.00	24	10.42	10.42	my_f_tell
0.00	6.00	0.00	6	41.66	258.14	getMethodFatHeaderFromStream
0.00	6.00	0.00				__fixdfsi
0.00	6.00	0.00				__muldi3
0.00	6.00	0.00	6	25.00	187.36	getLocalsBlobSignatureFromStream
0.00	6.00	0.00				_malloc_r
0.00	6.00	0.00	1	112.49	112.49	initStack
0.00	6.00	0.00				__divsi3
0.00	6.00	0.00				puts_r
0.00	6.00	0.00	1	99.99	4757.72	main2
0.00	6.01	0.00				_exception_handler
0.00	6.01	0.00				_printf_r
0.00	6.01	0.00	6	8.33	333.06	getMethodDefEntryFromStream
0.00	6.01	0.00				_sinit
0.00	6.01	0.00				localeconv
0.00	6.01	0.00				__smakebuf


```

0.00      6.01      0.00      _calloc_r
0.00      6.01      0.00      setlocale
0.00      6.01      0.00      6      0.00  248.23  getMethodSignature
0.00      6.01      0.00      1      0.00 49994.71  execute

```

Figure 5.7: Snapshot of the execution time of Co-designed Solution

A flat profiling is done as explained with and without the hardware Implementation and the results are tabulated. These are shown in figure 5.8.

$$\text{Speed Up Factor} = \frac{\text{Execution time}_{\text{Pure Software}}}{\text{Execution Time}_{\text{Co-designed}}}$$

Program	Software (microseconds)	Hardware (microseconds)	SpeedUp acheived	Comment
P1	38012.12	22360.07	1.7	Load args and load local vars intensive
P2	18032.21	7360.08	2.45	Load constants and arith operations intensive
P3	7836.23	4124.33	1.9	A simple function call
P4	10467.20	4757.72	2.2	Binary and unary operations intensive
P5	1043.10	869.16	1.2	Hello world program

Figure 5.8: Profiling results

From the above figures, the average speed up factor is found to be 1.89. Thus it can be concluded that the performance of a co-designed .NET processor is better than a pure software solution. This vindicates our aim that a hardware implementation is better than a pure software approach.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Although technologies like Java and .NET have become predominant day by day due to the increase in demand for homogeneous computing, their performance, due to interpretation, becomes inferior. It, therefore, becomes imperative to improve the performance. The .NET technology is a more desirable solution than Java because of its language interoperability. Use of FPGA reduces the design costs since they are cheaper than ASICs in addition to providing reconfigurability. This work concerns with improving the performance of .NET systems and the results show that a hardware implementation on an FPGA, as had been discussed in this thesis, indeed accelerates those programs that are intended to run on the .NET framework.

Future enhancements can be done in 4 stages to make the existing design perform better. Firstly object modeling instructions as specified in the .NET specifications [10] can be implemented in the software. This can be further extended so that even these object modeling instructions can be implemented in hardware. In this case the subset of object modeling instructions must be chosen carefully such that it doesn't produce too complex a hardware that may become a bottleneck in itself. The design can then be extended to perform better for specific embedded systems by optimizing specific features of the design such as the stack cache and extending the functional units to perform an extended set of operations specific to an embedded system. Finally, to make the system work consistently, a spill-fill handler must be added to handle stack overflow conditions.

CHAPTER 7

REFERENCES

[1] Erik Meijer and Jim Miller, 'Technical Overview of the Common Language Runtime' Microsoft, 2001

[2] An introduction to the Pico Java processor - Available at : <http://www.pages.drexel.edu/~ooo22/KImages/picoRep.htm>, 2002.

[3] Java Processors – available at www.elecdesign.com/Articles/ArticleID/3500/3500.html

[4] L.V. Nagendra Kumar, International Institute of Information Technology, Gachibowli, Hyderabad, India, 'JVM Implementation in FPGAs', B.Tech final year Project report, 2002.

[5] K John Gough, 'Parameter Passing for the Java Virtual Machine' Australian Computer Science Conference ACSC2000, Canberra, February 2000, IEEE Press.

[6] K John Gough and Diane Corney, 'Evaluating the Java Virtual Machine as a target for Languages Other than Java' Joint Modula Languages Conference JMLC2000, Zurich, September 2000

[7] Hejun Ma, Ken Kent, David Luke, 'An Implementation of the Hardware Partition in a Software/Hardware Co-Designed Java Virtual Machine', In the proceedings of IEEE on May 2004, Volume 4, ISSN:0840-7789 ISBN:0-7803-8253-6.

[8] The following article from the Intel website was referred.

"Itanium® Processor Family Performance Advantages: Register Stack Architecture", By Scott Townsend,

<http://www.intel.com/cd/ids/developer/asmo-na/eng/affiliate/hp/20314.htm>

[9] Plataforma.NET-available at 'http://people.ac.upc.edu/enric/PFC/Plataforma.NET/p.net.html', 2002 Topics Referred in this URL - Objectives and Future Projects.

[10] ECMA Draft (ECMA/TC39TG3/2000/3)- Part 3 IL Instruction Set, March 2000.

[11] K John Gough, 'Stacking them up: a Comparison of Virtual Machines', In the proceedings ACSAC-2001.

[12] www.xilinx.com/xup/mb_ref_guide.pdf, Xilinx Inc

[13] ECMA Draft (ECMA/TC39TG3/2000/3)- Part 3 IL Instruction Set, March 2000

<http://www.ecma-international.org/publications/standards/Ecma-335.htm>

[14]“Metadata Tables”- Vijay Mukhi, BPB Publications, ISBN 81-7656-605-5, 2004
<http://www.vijaymukhi.com/documents/books/metadata/contents.htm>

[15] Java Virtual Machine – available at <http://java.sun.com>

[16] Embedded System Tools Reference Manual, Embedded Development Kit EDK (8.1i), 2005.http://www.xilinx.com/ise/embedded/est_rm.pdf

[17] Ryan Rakvic, Ed Grochowski, Bryan Black, Murali Annavaram, Trung Diep, and John P. Shen. Performance Advantage of the Register Stack in Intel® Itanium™ Processors, Microprocessor Research, Intel Labs (MRL),2002.

A. APPENDIX

A.1: THE XILINX EMBEDDED DEVELOPMENT KIT

EDK is a series of software tools for designing embedded processor systems on programmable logic, and supports the IBM PowerPC™ hard processor core and the Xilinx® MicroBlaze™ soft processor core. Platform Studio™ is the graphical user interface technology that integrates all of the processes from design entry to design debug and verification.

The Embedded Development Kit is distributed as a single media installable CD image.

The components of the Xilinx® EDK are:

- Hardware IP for the Xilinx embedded processors and its peripherals
- Drivers, Libraries and a MicroKernel for Embedded Software Development
- Platform Studio tools
- Software Development Kit (Eclipse Based IDE)
- GNU Compiler and Debugger for C development for MicroBlaze™ and PowerPC™
- Documentation
- Sample projects

The version used for the project is EDK 7.1.

A.2: THE XILINX PLATFORM STUDIO (XPS)

XPS is an integrated design environment (IDE) used to develop EDK-based system designs. To generate a simple hardware system for EDK-based designs using Xilinx® EDK 7.1 and Xilinx® ISE™ 7.1i EDK hardware involves assembling a system that contains a processor along with buses and peripherals, generating an HDL netlist, and implementing the design using ISE implementation tools to generate a bitstream.

A snapshot of XPS is shown in Figure A1.

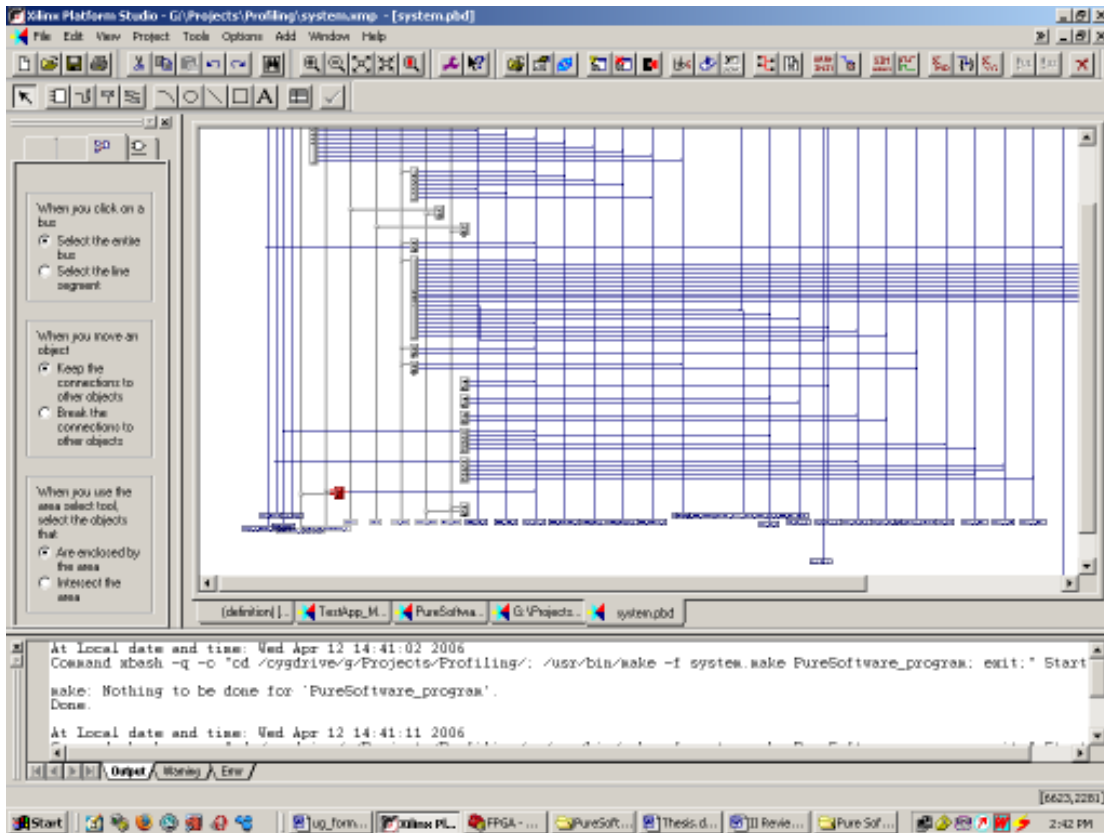


Figure A1: XPS in action

DESIGN FLOW IN XPS

The steps involved in creating a hardware system for EDK using XPS are as follows:

1. Create a New XPS Project
2. Select a Target Board
3. Select the Processor to be Used
4. Configure the Processor
5. Configure IO Interfaces
6. Specify Internal Peripheral Settings
7. Specify Software Configuration
8. View System Summary and Generate
9. View Peripherals and Bus Settings
10. Generate Bitstream
11. Download Bitstream and Execute

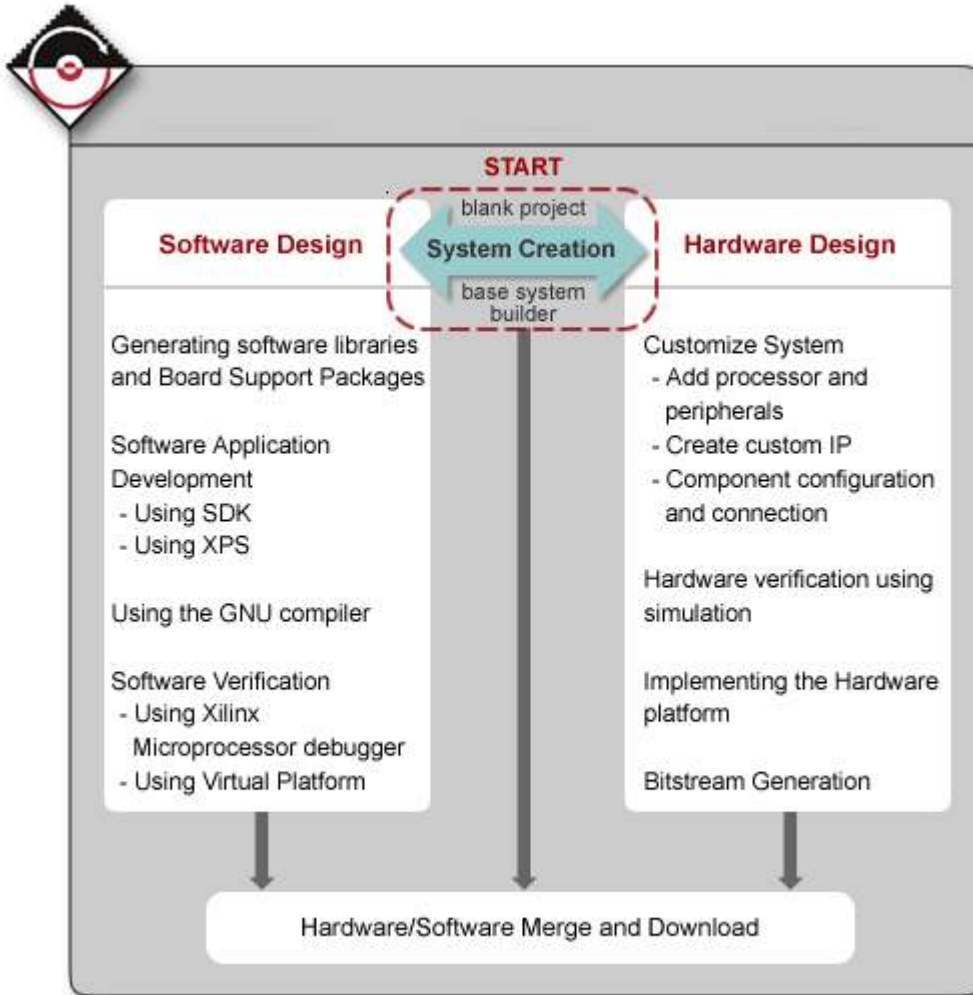


Figure A2: Design Flow in XPS

A.3: XILINX MICROPROCESSOR DEBUGGER (XMD)

The Xilinx® Microprocessor Debugger (XMD) is a tool that facilitates debugging programs and verifying systems using the PowerPC™ 405GP (Virtex™-II Pro & Virtex™-4) or MicroBlaze™ microprocessors. You can use it to debug programs running on a hardware board, Cycle-accurate Instruction Set Simulator (ISS), or MicroBlaze Cycle-accurate Virtual Platform (VP) system. XMD provides a Tool Command Language (Tcl) interface. This interface can be used for command line control and debugging of the target as well as for running complex verification test scripts to test a complete system.

```

C:\EDK\bin\nt\xmd.exe
Connected to the JTAG MicroProcessor Debug Module (MDM)
No of processors = 1

MicroBlaze Processor 1 Configuration :
-----
Version.....4.00.a
No of PC Breakpoints.....2
No of Read Addr/Data Watchpoints..1
No of Write Addr/Data Watchpoints..1
Instruction Cache Support.....off
Data Cache Support.....off
Exceptions Support.....off
FPU Support.....off
FSL DCache Support.....off
FSL ICache Support.....off
Hard Divider Support.....off
Hard Multiplier Support.....on
Barrel Shifter Support.....off
MSR clr/set Instruction Support...off
Compare Instruction Support.....off
Number of FSL ports.....1
JTAG MDM Connected to MicroBlaze 1
Connected to "mb" target, id = 0
Starting GDB server for "mb" target (id = 0) at TCP port no 1234
XMD%

```

Figure A3: XMD Command Shell

XMD supports GNU Debugger (GDB) Remote TCP protocol to control debugging of a target. Some graphical debuggers use this interface for debugging, including PowerPC and MicroBlaze GDB (powerpc-eabi-gdb and mb-gdb) and Platform Studio™ SDK (Eclipse based Software IDE). In either case, the debugger connects to XMD running on the same computer or on a remote computer on the Network. XMD reads XMP, MHS, and MSS system files to better understand the hardware system on which the program is debugged. The information is used to perform memory range tests, determine MicroBlaze to Microprocessor Debug Module (MDM) connectivity for faster download speeds and other system actions. Figure A4 describes the possible configurations of XMD.

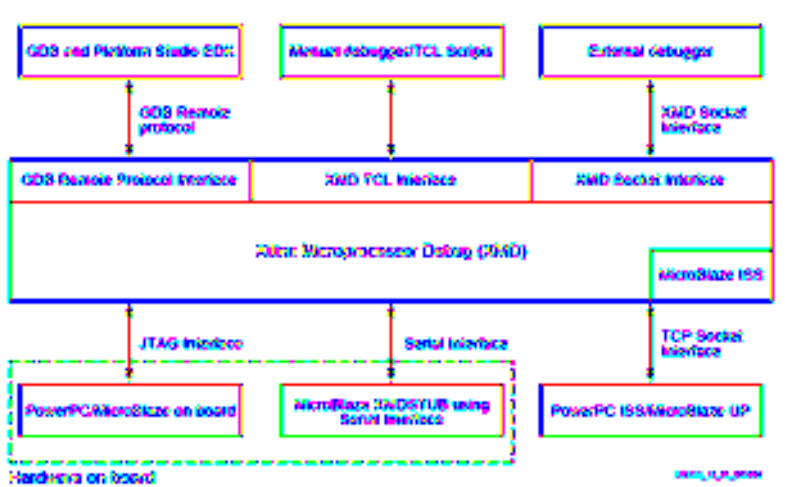


Figure A4: XMD Targets

A.4: DOCUMENTS THAT WERE USED AS A REFERENCE IN THE PROJECT

A.4.1: THE PLATFORM STUDIO USER GUIDE

EDK User Guide is a good place for first time users of EDK to understand the various design/debugging flows. Details about this document:

- ❖ Describes basic example designs for MicroBlaze and PowerPC. The doc covers both designing Hardware system as well as the Software Design.
- ❖ Usage of Xilinx MicroKernel and its components such as LibXilNet, LibXilMFS is discussed in the document.
- ❖ Using the Xilinx Microprocessor Debugger (XMD) for downloading program, profiling and creation of SystemAce files is explained with examples.
- ❖ Simulating a design using EDK tools and IP is discussed in the Simulation chapter
- ❖ EDK Software design flow is described in detail.

A.4.2: MICROBLAZE REFERENCE GUIDE

This document provides information about the 32-bit soft processor, MicroBlaze, included in the Embedded Processor Development Kit (EDK). The document is meant as a guide to the MicroBlaze hardware and software architecture.

This manual discusses the following topics specific to MicroBlaze soft processor:

- ❖ Core Architecture
- ❖ Bus Interfaces and Endianness
- ❖ Application Binary Interface
- ❖ Instruction Set Architecture

A.4.3: TOOLS AND IP REFERENCE GUIDES

Reference guides include detailed documents which explain

- ❖ Tool options and capabilities.
- ❖ Library and driver API and usage.

- ❖ Processor IP Datasheets
- ❖ User Core Templates usage and example systems.

A.4.4: EMBEDDED SYSTEMS TOOLS GUIDE

- ❖ Describes the Embedded Software Tools (EST) flow.
- ❖ Describes all the tools provided with EDK such as Library Generator, Platform Generator, GNU compiler framework, GNU debugger, Xilinx MicroProcessor Debugger, Simulation Generator and the Xilinx Platform Studio.

A.4.5: PLATFORM SPECIFICATION FORMAT

- ❖ Describes the Microprocessor Hardware Specification (MHS) format and the Platform Generator infrastructure for embedded processor peripheral definitions: the Microprocessor Peripheral Definition (MPD) format.
- ❖ Describes the MicroProcessor Driver Definition (MDD), MicroProcessor Library Definition (MLD) and Microprocessor Software Specification (MSS) format.
- ❖ OS and Libraries Reference Guide
- ❖ Describes the software libraries available for Xilinx Embedded Processors. The libraries include the Xilinx C library (libXil), the math library (libm), the Xilinx file support functions (libXil File), the Xilinx memory mapped file system (libXil MFS), the Xilinx networking support (libXil Net), the Xilinx device drivers (libXil Driver) and the Xilinx Standalone Board Support Package (BSP).

A.4.6: DRIVER REFERENCE GUIDE

- ❖ The libraries reference guide describes the overall philosophy and how the drivers can be hooked up in EDK designs. The drivers reference guide describes each driver delivered by the Embedded Development Kit in complete details.

A.4.7: PROCESSOR IP REFERENCE GUIDE

- ❖ Describes the usage of On-chip Peripheral Bus (OPB) and the IBM Processor Local Bus (PLB) is used in Xilinx FPGAs.
- ❖ Provides the design specification for all the processor IP, provided with the EDK.

A.5 INTERNALS OF THE .NET ARCHITECTURE

INTRODUCTION

The Common Language Environment (CLR) is the run-time environment of the .NET Common Language Infrastructure (CLI). It manages the execution of code and provides services that make the development process easier. The intermediate code form of the .NET system is called Common Intermediate Language (CIL) or the Microsoft Intermediate language (MSIL). As other virtual machines, the CLR too is stack-based machine.

THE ARCHITECTURE

The .NET CLR is a stack based machine. All operations are done on the stack. The local variables, the arguments that are passed back and forth between functions, the temporary evaluations etc. all reside on the stack. The .NET CLR is a complex machine and it provides a lot of high-level abstractions as primitives. For example, there is support of events, classes, arrays, delegates etc. even at the MSIL level. The CLR manages to provide such good features, thanks to the heavy amount of metadata that gets packed with a .NET executable.

THE COMPILATION PROCESS

.NET boasts of the feature of language interoperability. This is accomplished by having a VM kind of a setup. All the HLL programs get translated to an intermediate form called a .NET assembly. An assembly can either be a DLL or an executable. The executable assemblies are Portable Executable (PE) files. The MSIL instructions are encoded in these assemblies, which are then interpreted by the CLR. A heavy amount of

metadata gets packed with an assembly, which is made use of by the CLR during the interpretation stage. The PE and the metadata play a pivotal role in the .NET framework and are explained in [1]. We now proceed to give a general overview of the PE format.

The Win32 PE

The file format for .NET executables is a strict extension of the current Portable Executable (PE) File Format. This extended PE format enables the operating system to recognize runtime images, accommodates code emitted as CIL or native code, and accommodates runtime metadata as an integral part of the emitted code. The PE format would require pages of explanation. Loads of documentation is already available. We describe here only those parts of the PE file that are relevant to the work.

The figure below provides a high-level view of the CLI file format. All runtime images contain the following:

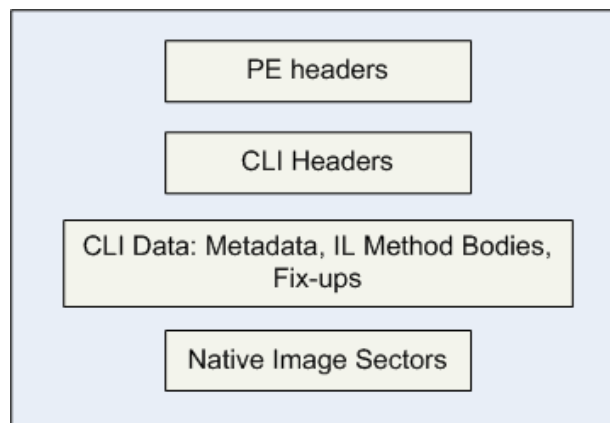


Figure A5: CLI File Format

- PE headers, with specific guidelines on how field values should be set in a runtime file.
- A CLI header that contains all of the runtime specific data entries.
- The sections that contain the actual data as described by the headers, including imports/exports, data, and code.

The PE optional header, which forms a part of the PE header, contains constructs called directories. The optional header data directories give the address and size of several tables that appear in the sections of the PE file. Each data directory entry contains the Relative Virtual Address (the offset of a data item from base address of the file) and

Size of the structure it describes, in that order. The CLI header is found using CLI Header directory entry in the PE header. The CLI header in turn contains the address and sizes of the runtime data i.e. for metadata and for CIL in the rest of the image.

METADATA – THE HEART OF A .NET ASSEMBLY

Many of .NET's high level and attractive features are made possible only because of the massive magnitude of the metadata that gets packed with a .NET assembly. The metadata is organised inside a .NET assembly in two forms: tables and heaps.

Heaps

There are five possible types of heaps:

- ❖ **String:** Metadata preserves name strings, as created by a compiler or code generator, unchanged. These strings are stored in this table.
- ❖ **Blob:** Contains indices for various table entries
- ❖ **UserString:** Contains the Unicode string constants used in the source file.
- ❖ **GUID: Globally Unique Identifier,** a 16-byte long number typically displayed using its hexadecimal encoding. While the other three heaps are byte arrays, this heap is an array of 16-bit entries.
- ❖ **#~ (The Tilde stream):** points to the physical representation of a set of tables.

Tables

Each entry in each column of each table is either a constant or an index. Constants are either literal values or bitmasks. Most bitmasks are 2 bytes wide but there are a few that are 4 bytes. Each index is either 2 or 4 bytes wide. The index points into the same or another table, or into one of the four heaps. The size of each index column in a table is only made 4 bytes if it needs to be for that particular module. So, if a particular column indexes a table, or tables, whose highest row number fits in a 2-byte value, the indexer column need only be 2 bytes wide. Conversely, for tables containing 64K or more rows, an indexer of that table will be 4 bytes wide. Indexes to tables begin at 1.