IMPLEMENTATION OF A JAVA PROCESSOR ON A FPGA

by

OMKAR JOSHI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2016

Acknowledgements

I would like to express my gratitude to the faculty at UT Arlington, my thesis committee Prof. David Levine, Dr. Ali Davoudi and Dr. Ramtin Madani, most particularly Prof. David Levine for the constant encouragement and guidance throughout, without whom this project would never have come to fulfilment.

I want to extend my thanks to Mr. William Hunter, for helping with the implementational details and Dr. Bill Carroll for his sound words of advice.

Last but not the least, I express my thanks to my parents, family and friends for supporting me and being there every step of the way.

Nov 17, 2016

Abstract

IMPLEMENTATION OF A JAVA MACHINE ON A FPGA

Omkar N Joshi, MS

The University of Texas at Arlington, 2016

Supervising Professor: David Levine and Ali Davoudi

Java, a programming language developed by Sun Microsystems in 1991, now managed by Oracle, has become one of the most popular computer languages for application development. This popularity can be credited to Java being architectural neutral and portable. It means that a Java program executed on any computer will yield the same result, irrespective of the underlying hardware.

When a Java program is compiled it creates a Java class file. The class file contains instructions known as Bytecodes, which are executed by the Java Virtual Machine (JVM). The JVM is an abstract processor, which interprets and translates the bytecodes into instructions for the native processor. The process of interpretation, along with functionality such as dynamic linking, Just-in-time compilation and on demand class loading, makes the execution of a Java application slower than compiled programs.

In order to speed up this execution of the Java program, this project has developed a processor for which the bytecodes are the native instructions. This eliminates the time spent on interpretation and translation. Also, with the implementation of the Java Machine, certain run-time dependencies can be eliminated by pre-processing the class file, before loading it into the memory of the processor.

By developing the processor on a Field Programmable Gate Array (FPGA), the Java Machine can be kept up to date with the newest Java standards even after it is installation in the field. The FPGA processor can also be optimized to specific applications by adding application specific hardware modules to speed up the processing.

TABLE OF CONTENTS

List of Illustrations

List of Tables

Chapter 1

Introduction to Programming Languages

Definitions

For computers to accept commands from a user and perform specific tasks, there needs to be a means for communication between the humans (users) and the computer. This means for communication are provided by programming languages.

Some definitions of a programming language are as follows:

- “A programming language is a notation for writing programs, which are specifications of a computation or algorithm” [1].
- “A vocabulary and set of grammatical rules for instructing a computer to perform specific tasks” [2].
- “A programming language is a computer language engineered to create a standard form of commands. These commands can be interpreted into a code understood by a machine. Programs are created through programming languages to control the behavior and output of a machine through accurate algorithms, similar to the human communication process” [3].
- “A programming language is considered to be a set of characters and rules for combining them which have the following characteristics: (1) machine code knowledge is unnecessary; (2) there is good potential for conversion to other computers; (3) there is an instruction explosion (from one to many); and (4) there is a notation which is closer to the original problem than assembly language would be” [4].

Classification of Programming Language

Programming Languages, depending on their nature of use and programmability, can be classified as below. This classification and description is inspired from [5]:

- Machine languages
- Assembly languages
- Middle-level languages
- High-level languages
- System languages
- Scripting languages
- Domain-specific languages
- Esoteric and Visual languages

Machine languages are programming languages that can be directly interpreted in hardware. For modern computers, the machine language is usually opcodes which is a stream of binary data. Assembly languages are machine languages wrapped in a text making it human readable. Assembly languages use mnemonics to differentiate instructions. These instructions can be then fed to an assembler to generate machine code. Assembly languages and machine-level languages together make up low-level languages. Middle-level languages, such as C/C++ can be used to program any computer but the results of these programs may wary depending on the under lying architecture.

High level languages, unlike machine level languages and assembly languages, can usually be portable from one machine to another. High level languages use either a compiler or an assembler to convert a high level, human readable code into machine code. Some examples of high level languages are Java, Pascal, Python, Ruby and C#.

System languages are languages such as PL-6 and ADA, which may be used more for managing the computer system rather than writing application code. They usually perform tasks such as memory management, device drivers, kernel and thread managements. Scripting languages are frequently used to write high level code, which connect one or more applications together. Examples of scripting languages are PHP, python. Domain-specific languages are languages which can only be used for certain applications only, HTML is one such language for web pages and shell scripting for Unix. Esoteric and visual languages are meant for educational and in some cases as a challenge. Example of such languages are Turtle Graphics and Scratch.

Virtual Machines

The term Virtual machine was first coined in the 1960s as an operating system concept. It can be described as a software abstraction that emulates like a computer system (real computer) which can operate independently [6]. The virtual machine acts as a target to for a compilation system or a programmer. Many of high level languages such as Java, C#, Pascal, Python etc. use a virtual machine for its application. A virtual machine like an actual processor has an instruction set and manipulates memory during runtime [7]. A comparison of some of the more popular virtual machines is listed below:

Virtual machine	Languages	Interpreter	JIT	Implementation language
Common Language Runtime (CLR)	C#, C++/CLI, F#, VB.NET	No	Yes	C#, C++

Adobe Flash Player	ActionScript, SWF (file format)	Yes	Yes	C++
DotGNU-Portable.NET	CLI languages including: C#	No	Yes	C, C#
Forth	Forth	Yes	No	Forth, Forth Assembler
HHVM	PHP, Hack	Yes	Yes	C++, OCaml
JVM	Java, Jython, Groovy, JRuby, C, C++, Clojure, Scala and several others	Yes	Yes	JDK, OpenJDK : Java, C, ASM ; IcedTea
LLVM	C, C++, Objective-C, Ada, Fortran, andRust	Yes	Yes	C++
Mono	CLI languages including: C#, VB.NET,Iron Python, IronRuby, and others	Yes	Yes	C#, C
p-code machine	Pascal			
Parrot	Perl (6 and 5), NQP-rx, PIR, PASM, PBC,BASIC, bc, C99, ECMAScript, Lisp, Lua,m4, Tcl, WMLScript, XML, and others	Yes	Yes	C, Perl
Perl virtual machine	Perl	Yes	No	C, Perl

PyPy	Python	Yes	Yes	Python
Rubinius	Ruby	Yes	Yes	C++, Ruby
Silverlight	C#, VB.NET	Yes	Yes	C++
SQLite	SQLite opcodes			
Squeak	Squeak Smalltalk	Yes	Cog and Exupery	Smalltalk/Slang
Zend Engine	PHP	Yes	No	C

Table 1-1 Comparison of Virtual Machines and features [8]

The JVM and the JVM bytecodes are not only used by JAVA, some other programming languages that generate Java bytecode are listed below [8]:

- Clojure, a functional Lisp dialect.
- Groovy, a programming and scripting language.
- Scala, an object-oriented and functional programming language.
- JRuby, an implementation of Ruby.
- Jython, an implementation of Python.

Chapter 2

Java

Introduction to Java

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It is now maintained and managed by Oracle.

Some of the features/characteristics of Java, as mentioned in the Oracles developers guide [9] are:

- Object Oriented:

  Java is an object-oriented language, and supports the object oriented technologies of C++ with certain modifications and enhancements. It is not purely object oriented because it supports primitive data types.

- Architecture Neutral and Portable:

  Source code written in Java is compiled into an architecture independent object code and stored in a class file. This class file can be compiled on a machine and later executed on another machine, and should work with no or little modifications. The object code is executed by on a Java Virtual Machine (JVM) for the under lying architecture. This makes it architecture neutral.

- Distributed:

  Java can be used to create distributed applications in the sense that programs can be written to access any file on the internet.

- Secure and Robust:

  The Java compiler and interpreter have an extensive error checking mechanism. The object code has several tests applied to check for illegal codes, illegal memory accesses, illegal data conversations and verified for opcode parameter types. Java

also manages dynamic memory and exceptions. This makes the Java code secure and robust.

- High Performance and Multi-Threading:

  Java comes built in with support for multi-threading. With multi-threading we can have operations executing in parallel, which improves the performance in terms of execution time.

- Dynamic:

  Linking of the various modules in Java takes place during runtime. This makes Java dynamic.

Terminology related to Java

To understand Java and its features completely, it is important to understand the terms related to the programming language:

- Source File and the Class File
- Java Virtual Machine
- Java Runtime Environment
- Java Development Kit

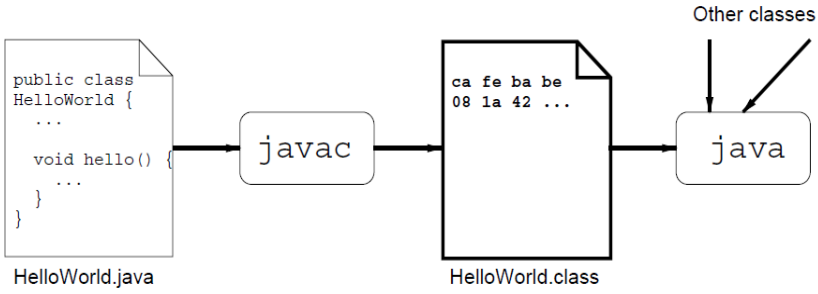


Figure 2-1 Compilation and execution of Java Classes [10]

Source File and the Class File

The file in which a Java program or class is written is the source file. These files are saved with a Java extension. When the source file is fed to the Java compiler the file generated is a class file. The class file has a .class extension and is the input to the JVM. It contains all the information necessary for Java to execute the program/class. We will look into details of the class file, in a later section.

Java Virtual Machine

The Java Virtual Machine (JVM) is an abstract machine, which executes the Java class file. The JVM is what enables a compute to execute the Java program. Oracle which maintains Java, does not specify an implementation of the JVM, rather defines a specification that allows class files to be executed. Some tasks performed by the JVM are, as follows, resolving dynamic dependencies, interpreting bytecode into native processor instructions, garbage collection, just-in-time compilation and many more.

The HotSpot is one implementation of JVM by Oracle. Based on the HotSpot implementation, BEASystems came out with its own JVM named JRocket. IBM J9 and Kaffe are similar JVM implementations which satisfy the Oracle specifications.

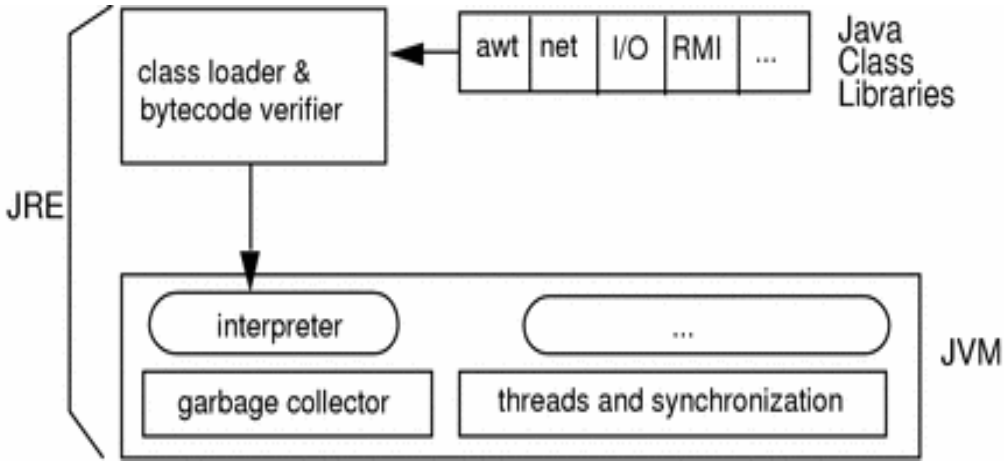


Figure 2-2 Components of JRE [9]

Java Runtime Environment

The Java Runtime Environment (JRE) provides all the functionality to run a Java program. The JRE is not capable of compiling a source file. It provides the libraries, the JVM and other such components necessary to run applets and applications written in Java. The two important sections of the JRE are the Java Plug-in, which allows Java applets to run in a browser, and Java Web Start which allows standalone applications to be deployed over a network [11].

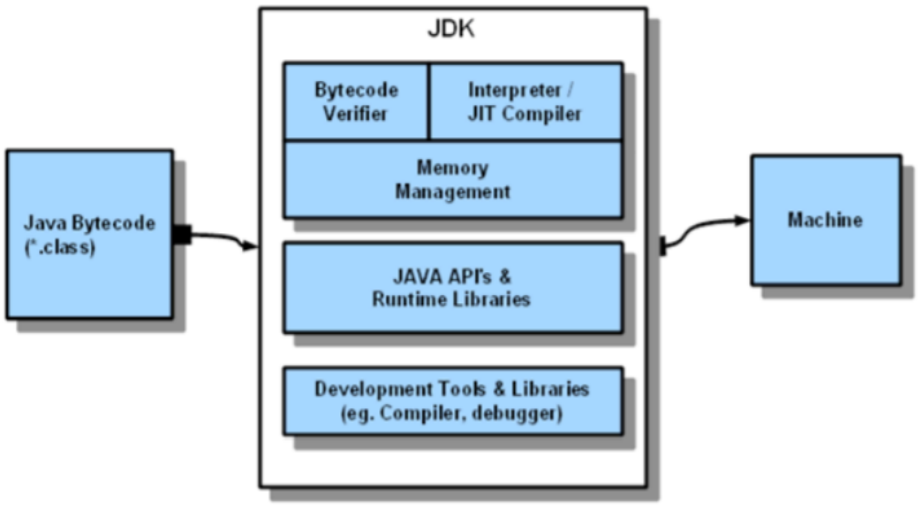


Figure 2-3 Components of the JDK [12]

Java Development Kit

The Java Development Kit (JDK) is the superset of the JRE. The JDK contains all the components of the JRE along with tools that allow a source file to be compiled and debuggers that allow for development of Java applets and applications.

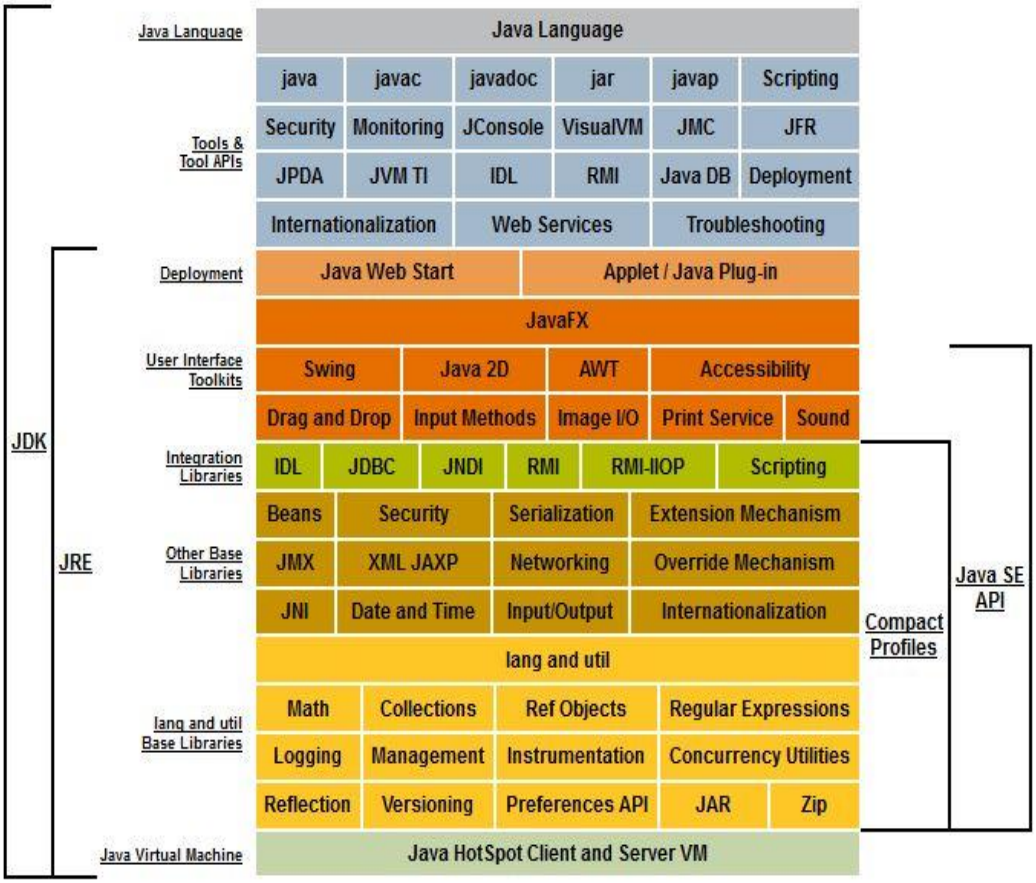


Figure 2-4 Components of Java platform by Oracle [11]

Chapter 3

Java Virtual Machine and the Class File

The Java Virtual Machine

The JVM, as mentioned previously, is an abstract (virtual) computing machine. The JVM is what makes Java independent of the underlying hardware and operating system on which the source code is running. It does so by providing a layer of abstraction between the compiled Java code and the underlying system.

When a Java program is compiled, it creates a class file. This class file contains the bytecode. The Java bytecode acts is the machine language for the JVM. Each JVM instruction contains a one-byte opcode and zero or more operands. Each JVM instruction, bytecode, has a pre-defined mnemonic and this can be the assembly language equivalent.

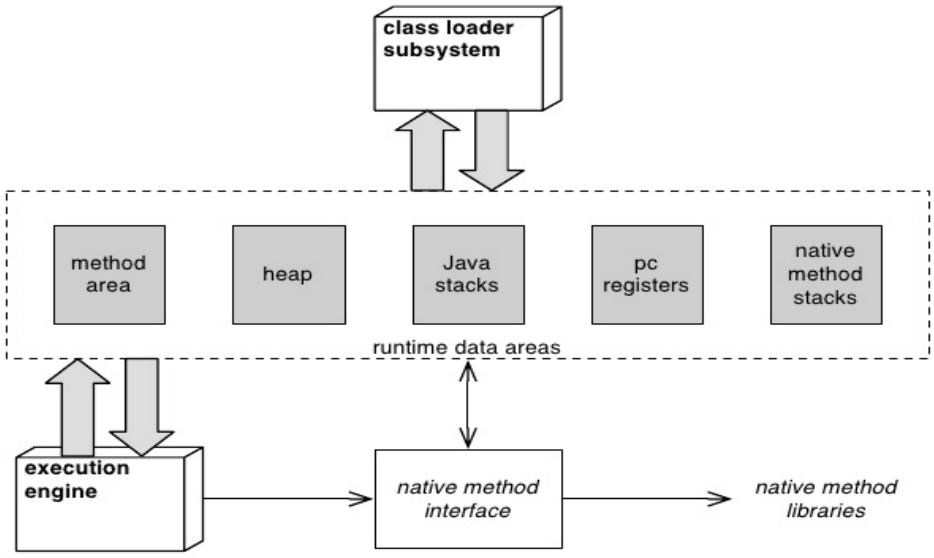


Figure 3-1 Internal Architecture of the JVM [13]

The JVM is a 32-bit stack based machine. This means the size of a word in the JVM is 32 bits i.e. each register in the stack can store a 32-bit value. The stack oriented design of the JVM helps keep the instruction set compact and the implementation small.

But the JVM also supports primitive data types: byte (8-bits), short (16-bits), int (32-bits), long (64-bits), float (32-bits), double (64-bits), and char (16-bits). All the numeric data types the JVM operates on are signed, except for char which is a 16-bit Unicode encoded character. The JVM can address up to 4 gigabytes (2 to the power 32) memory addresses. The location of these 4GB is the decision of the implementer of the particular JVM.

The JVM can be split into three sections:

- Class Loader Sub-system
- Runtime Data Areas
- Execution Engine

Class Loader Sub-system

The class loader sub-system, as the name indicates, loads the JVM memory with the relevant class information. The class loading occurs during run time and is on demand. The operation of the class loader subsystem can be divided into three sections (a) load, (b) link and (c) initialize [13].

During the load phase, the class loading subsystems reads the class files from different sources such as the filesystem directly, a .jar file or the network socket and the Internet. After reading these files, they are loaded into the appropriate memory area of the JVM. The class loader has three parts itself, which are (a)bootstrap class loader, (b) extension class loader and the (c) application class loader.

The bootstrap loader is responsible for loading the classes internal to the JVM implementation. These classes are usually located in the rt.jar folder which is downloaded as a part of the JRE. The extension class loader also loads file provided by the JRE, and these are additional classes used by the Java application present in the jre/lib/ext folder. The application class loader loads the class files generated for the specific application.

These classes are mentioned as part of the environment class path or part of the -cp parameter, which is part of the JVM.

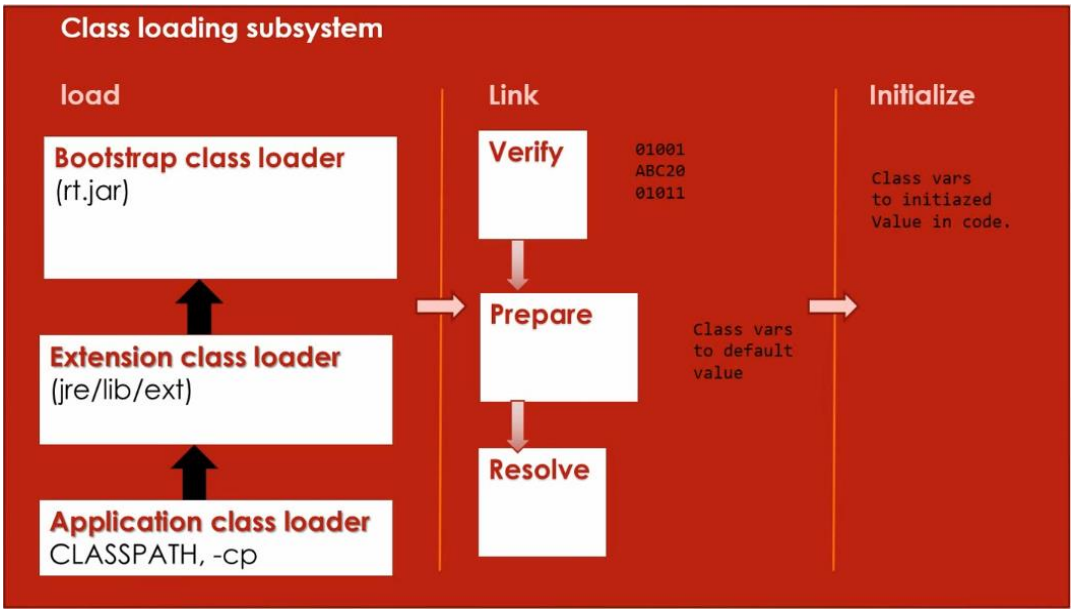


Figure 3-2 Components of the Class Loading Subsystem [14]

After the load phase, the link phase is involved. The operation of the link phase is split into three sections (a) verify, (b) resolve and (c) prepare. These sections may be run in parallel or sequentially depending on the implementation of the JVM.

During the verify phase, the class loader checks if the class file loaded by the loader is compatible with the version of the JVM. It performs bytecode verification, magic number checks and checks if the interface is structurally correct. During the prepare phase, static fields are assigned to the classes and interfaces, the memory allocation for instance variables is done during run-time. These static fields are then initialized to their default values, specified by the Java specifications. During the resolve phase, the symbolic references to different elements such as arrays, fields and static and dynamic classes are translated from constant pool references to actual references in the memory. Each kind of reference resolution has its own format and is a heavy process.

The final phase of the class loader subsystem is initialize. During this phase, static initializers of the class are executed, along with setting the initial values of static fields.

Runtime Data Areas

The JVM, to execute a program, needs memory not only to store the class file and its related components but also to store data generated during the execution such as objects, arrays, local variables, return values and intermediator results. The memory which the JVM accesses is known as the run time data areas. For the JVM, these run time data areas can be divided into 5 areas: (a) the method area; (b) heap (c) Java Stack; (d) Program Counter (PC) registers; (e) native method stack.

- Method Area and the Program Counter(PC):

  The Method area is where the bytecode resides in the JVM. The PC points to the next instruction to be executed. Each thread has its own PC and is used to keep track of the execution.

- Heap:

  The heap in Java is where all the objects are stored. When a memory is allocated using the keyword `new`, the same amount of memory is kept aside in the heap for the particular object. The Garbage Collector runs on the heap, and frees up memory when the object can no longer be accessed i.e. is out of scope.

- Java Stack:

  The Java stack not only contains the local stack but also the local variables array. Each method in the code has its own stack and is called a stack frame. The stack is used to store parameters for the bytecode instructions along with the results and hence, acts as the workspace for the instructions. They are also used to pass parameters to and from (return) different methods.

The local variables array contains all the variables used by the method. The local variables array is pointed to by the vars register.

The stack frame also consists of another section called as the execution environment. The execution environment contains registers such as the optop register which points to the top of the operand stack. It execution environment is itself pointed to by the frame register.

- PC registers and native method stack:

  The PC register, like in any other processor points to a memory location in the method area, from where the next instruction is to be fetched. With multi-threading, each thread is assigned its own PC, which allows to keep track of the individual instruction to be executed in each thread.

  The native method stack is used in case of methods or functions imported into the Java program from other languages such as C/C++.

Execution Engine

The execution engine, as the name indicates, is responsible for execution of the Java bytecode. The execution engine can be broken down into 4 sections, which perform different operations but all run in parallel. The four blocks are (a) interpreter, (b) JIT compiler, (c) hotspot profiler and (d) garbage collector.

The interpreter reads the Java bytecode from the runtime data area and converts them into the native processor instruction. This translation of instruction sets is what allows Java to be an architecturally neutral programming language. To do this translation, the interpreter uses the native method libraries present in the JVM. The JIT compiler speeds up the translation of the instructions by doing them in advance. The execution engine initially executes the interpreter, and at the appropriate time invokes the JIT compiler to translate most of the bytecode into native instructions for the processor. This may be a

memory heavy process; it leads to considerable speed up in running the Java application. Similar optimization is provided by the hotspot profiler; it detects a sequence of bytecodes repeatedly used and stores them in memory which is faster to access. This saves memory read times further boosting the performance of the JVM.

The garbage collector is responsible for scavenging process. It frees up the memory used by the heap. When an object goes out of bounds, the garbage collector frees up the memory by marking it as such. Such processes in the other languages must be performed by the user by using the free keyword. And may lead to data corruption and/or add to the memory usage by the program.

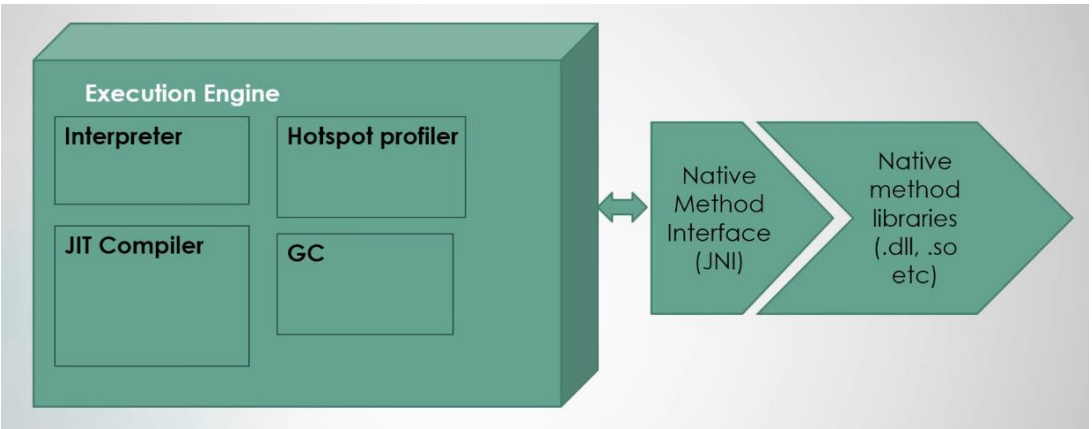


Figure 3-3 Components of Execution Engine [14]

The Class File

The class file is a file with the extension .class and is generated by the Java compiler. It contains the bytecode and can be executed by the JVM. The Java compiler will generate a class file for each class or interface defined in the source code. The class file is organized as a stream of 8-bit bytes. All the 16-bit, 32-bit and 64-bit values are constructed by simultaneously reading multiple bytes. The bytes are represented in big-endian notation i.e. the most significant byte comes first.

For ease of understanding, the oracle documentation defines its own set of datatypes: The types u1, u2, and u4 represent an unsigned one-, two-, or four-byte quantity, respectively.

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figure 3-4 Structure of the Java class file [15]

The different sections of the class file are as follows:

- `magic`

  The magic is a 4-byte code used to identify a class file. The value of magic has to be 0xCAFEBABE.

- `minor_version and major_version`

  The major and minor are both 2-byte fields and are version numbers of the respective class file. Together they determine the version of the class file format.

- `constant_pool_count`

  The constant_pool_count field gives the number of entries in the constant pool plus one.

- `constant_pool[]`

  The constant_pool is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The format of each constant_pool table entry is indicated by its first "tag" byte. The constant pool occupies about 60% of the class file and is the biggest section of the class file.

- `access_flags`

  The access_flags field is used to denote the permissions and properties of the respective class or interface.

- `this_class`

  The this_class field points to an entry of the type CONSTANT_Class_info in the constant pool table. It represents the class or interface defined in this class file.

- `super_class`

  The super_class field either is a zero or points to an entry of type CONSTANT_Class_info in the constant pool table. It represents the direct super class of the class or interface defined in the class file.

- `interfaces_count`

  The value of the interfaces_count item gives the number of direct superinterfaces of this class or interface type.

- `interfaces[]`

  Each value in the interfaces array points to an entry of type CONSTANT_Class_info in the constant_pool table. It representing an interface that is a direct superinterface of this class or interface type.

- `fields_count`

  The fields_count field gives the number of field_info structures defined in the class file.

- `fields[]`

  A field in Java is defined as a class, interface, or enum with an associated value. Fields are variables defined in the class and have class wide scope, unlike local variables.

  The entries in the fields table give the complete description of the fields declared in this class or interface.

- `methods_count`

  The method_count field gives the number of method_info structures defined in the class file.

- `methods[]`

  A method in Java can be defined as a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name. Methods are equivalent to functions in C/C++.

  The entries in the method table give the complete description of the method declared in this class or interface.

- `attributes_count`

  The attributes _count field gives the number of attributes_info structures defined in the class file.

- `attributes[]`

  Attributes are used in the class file, `field_info`, `method_info`, and `code_attribute` structures of the class file format. The entries in the attributes table give the complete description of the attributes declared in this class or interface.

Bytecode

As described earlier, the JVM is an abstract processor, and similar to any other physical processor, it has its own instruction set known as the Java bytecodes. The bytecodes are present in the `code_attribute` section of the class file, in the `method_info` structure. These bytecode instructions are then interpreted into the native processor instructions by the interpreter.

The bytecodes are a sequence of instructions to the JVM, that allows it to perform a specific task defined in the corresponding method. Each bytecode instruction has an 8-bit opcode and may have additional bytes. Most bytecode instructions have a pre-defined fixed length, some instruction lengths may vary and must be decoded during run-time. The opcode in each instruction specifies the operation to be performed, while any additional information required is available in the additional bytes.

Since the length of the opcode is 8-bit, the JVM allows 256 instructions. As of 2015 (Java 8), only 198 instructions are in use, 3 opcodes are reserved from use permanently, while the remaining 54 can be used in the future [16]. The opcodes 254 and 255 i.e. 0xfe and 0xff are reserved to allow implementation of traps or backdoors. These traps are meant to allow special functionality to be implemented in both hardware and software. These instructions are labeled impdep1 and impdep2 respectively. The third reserved opcode is 202 i.e. 0xca, which is labeled as breakpoint and is intended to be used for debugging [16].

Prefix-suffix	Operand type
i	integer
l	long
s	short
b	byte
c	character
f	float
d	double
z	boolean
A	reference

Table 3-1 List of prefix/suffix and the datatype represented [16]

To add to ease of understanding the Java bytecode instructions, the mnemonics contain prefixes or suffixes which specify the datatype of the operands along with the operation. The suffix and corresponding datatype can be seen in table 3-1.

Similar to any other assembly instruction set, the Java bytecode instructions can be classified, based on their operation, into the following categories, as mentioned in [17]:

- Operand stack management
- Arithmetic and logic
- Control transfer
- Load and store

- Object creation and Field Access
- Method invocation and return
- Type conversion
- Special Instruction

Operand Stack Management Instructions

Instructions of this type can be used to push values onto the stack top from different sources. The instruction of type ldc is used to push constants onto the stack from the constant pool, while the instructions bipush and sipush push constants onto the stack from the method area. The const instructions are used to push constants of certain commonly used values with different datatypes onto the stack.

Arithmetic and Logical Instructions

The JVM is capable of handling arithmetic operations such as add, subtract, multiply, divide and modulo; and logical operations such as logical AND, logical OR, logical XOR, arithmetic and logical shifts. Each of these operation for the primitive datatypes is handled by the JVM, and thus has its own instruction. For example, instruction fadd adds two floats values from the top of stack. Similarly, instruction iushr right shifts the top of stack by a specified number of times.

Control Transfer Instructions

In high level languages, code snippets such as for loops, if-else blocks and do-while loops generate code which may need to transfer control or update the PC to a non-incremental value. The same functionality in bytecode is provided by instructions such as goto and of type if_compare. The goto in this case, is an unconditional jump to a different location. The if_compare set of instructions compare values of two operands, and accordingly decide if the branch is to be taken or not.

Load and Store Instructions

Load and store instruction are used to move, operands from the top of stack to the local variables frame or array elements. The load instructions push the value onto the stack top, whereas the store instructions pop values from the stack top and save them in the array or local variables.

Similar to arithmetic and logical instructions, each datatype and destination has its own instruction. For example, the instruction lastore stores a long integer into an array. Similarly, iload_1 copies an integer from the local variable 1 onto the stack top.

Object Creation and Field Access Instructions

Instructions with the new keyword are usually instruction used for instantiation of class i.e. object creation. These instructions usually assign memory for objects on the heap. Examples of such instructions are anewarray. Instructions putfield and setfield are used to update or read the value of static fields, respectively.

Method Invocation and Return Instructions

Methods in Java can have different access specifiers such as static, private and virtual. Depending on the type of method, instructions such as invokestatic and invokevirtual are used, to invoke methods from a superclass invokespecial instruction is used. Programmers can return either a void, a reference or even numeric values such as integers and floats. To facilitate proper values to be transferred back to the calling method, instructions like areturn, dreturn, ireturn and return and many more are available.

Type Conversion Instruction

Casting allows the programmer to change the primitive type, to either round up or to obtain higher accuracy. The same operation in assembly instructions is performed by type conversation instructions. Java allows to increase or decrease the resolution of a

variable, and even integer to decimal and vice versa conversions. Some type conversion instructions are f2i, i2l, i2d and i2c.

Special Instructions

Tasks such as exception throwing, synchronization and table look ups are also assigned instructions, to allow the tasks to be performed internal to the JVM. Examples of such instructions are monitorenter, monitorexit, lookupswitch and tableswitch.

Chapter 4

Programmable Hardware

Introduction to Programmable Hardware

Any hardware device which can be programmed to perform a specific task is called as programmable hardware. Many of these devices can be programmed in the field after their installation and thus are called field programmable. These devices are generically called Programmable Logic Device (PLDs). Unlike most hardware chips that have a predefined function when they are manufactured, PLDs have an undefined function at the time of manufacture. Before PLDs can be used in a circuit, they need to be programmed and assigned a specific function to be written.

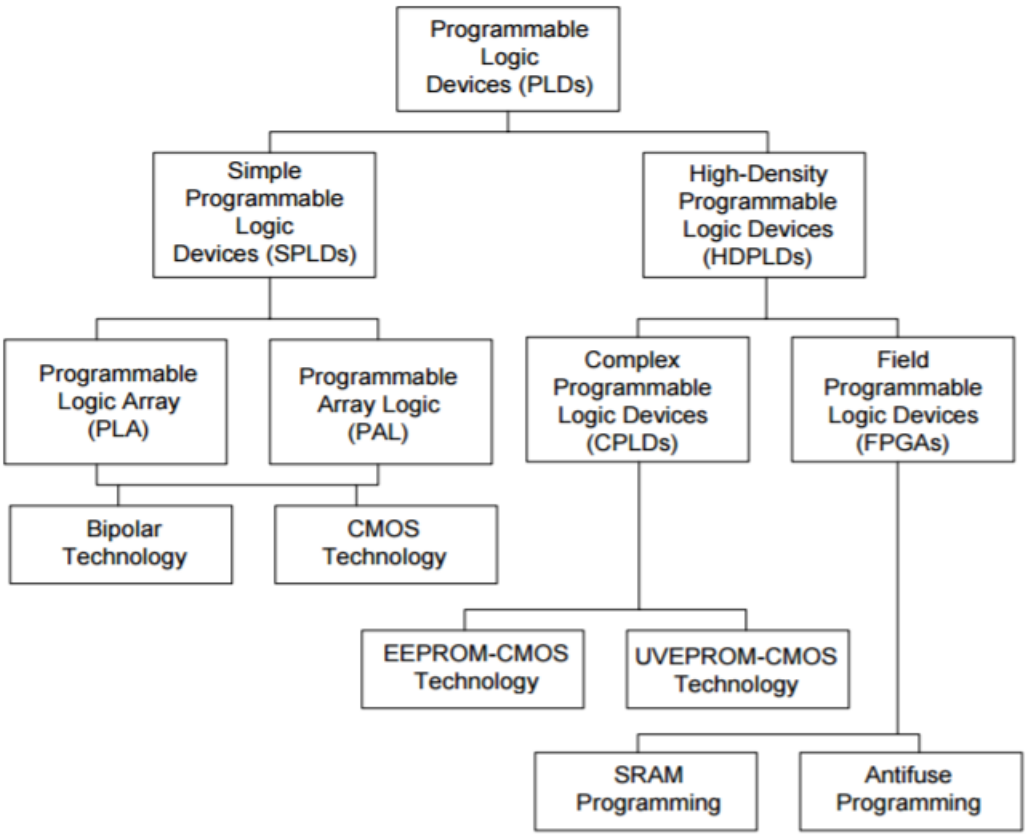


Figure 4-1 Classification of Programmable Hardware [18]

The figure above shows the classification of different Programmable Logic Devices. The devices categorized under SPLD were devices that came out earlier and have very limited programmability, and hence HDPLDs were developed or evolved from SPLDs.

Programmable ROM

Though programmable ROM is only memory, engineers created with a way to implement logic with them. The address lines of the ROM were used as inputs and the data lines acted as the output. But since, most logical functions needed more than a few product terms, the use of PROMs for realization of logical circuits was highly inefficient.

Programmable Logic Arrays

PLAs also known as Field Programmable Logic Arrays (FPLA) were the first hardware devices designed specifically as programmable hardware. They were introduced by Texas Instruments in 1970 [19]. The PLA architecture consisted of two layers, the first was the plane of programmable AND gates and the second, a plane of programmable OR gates. These PLAs were designed in such a way that the inputs (or their compliment) could be ANDed together. And the output of the AND plane would be routed to the OR plane. This made PLAs suitable for implementing sum-of-product logic functions.

The drawbacks of PLAs were slow performance due to high propagation delays and tedious programming technique.

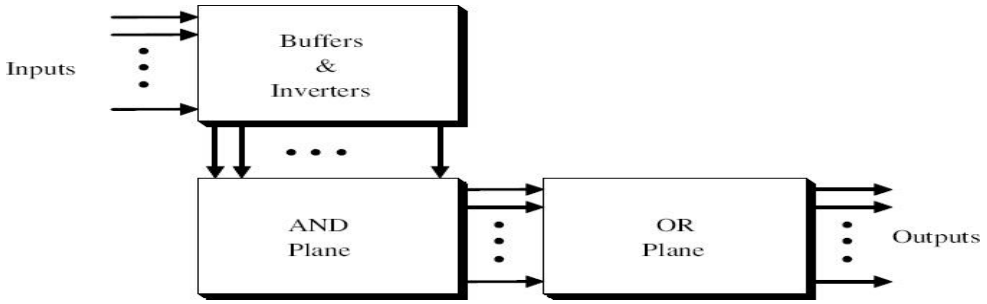


Figure 4-2 Internal Architecture of PLAs [20]

Programmable Array Logic

To overcome the drawbacks of PLAs, Programmable Array Logic (PAL) were developed. Instead of having two levels of programmable gates, the PALs had only had a single plane of programmable AND gates, that fed their output to fixed OR gates. The inputs of the PALs were connected to the AND gates via a buffer and inverter block. The output of the OR gates in PALs were connected to flip-flops allowing developers to implement sequential circuits.

As the PALs had a fixed OR gates, the flexibility of these devices were adversely affected. To overcome this drawback manufacturers created many variants of PALs.

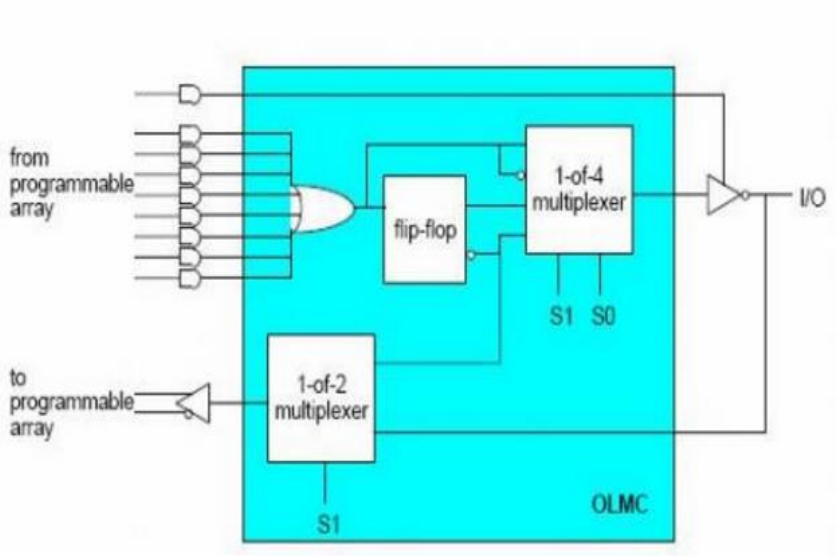


Figure 4-3 Internal Structure of Generic Array Logic [21]

Generic Array Logic

Generic Array Logic (GAL) devices were developed with much higher gate density and hence could replace more than one PAL or PLA. But the main advantage of GALs over the earlier generations was programmability. This allowed GALs to be used for prototyping and allowed design changes to be made without alterations to the circuit.

Complex Programmable Logic Devices

Due to advancement in fabrication techniques it was possible to integrate multiple SPLDs such as GALs and PLAs into a single chip. Such devices were called Complex Programmable Logic Devices (CPLD). CPLDs are a combination of a bank of macrocells and fully programmable AND and OR gates. The AND/OR gates are used to implement the different logic functions, while, the macrocells are used to provide both sequential and combinational logic along with feedback paths.

Along with the programmable gates and the macrocells, CPLDs contain a Global Interconnect Matrix which allows the different macrocells to be connected to each other and the I/O pins. Most CPLDs contain non-volatile memories which allow the device to be powered down yet maintain its configuration.

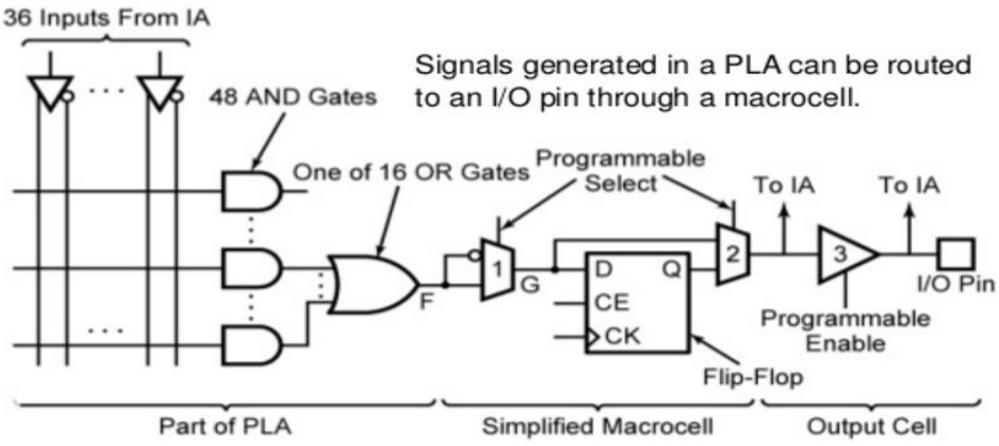


Figure 4-4 Functional Block diagram of an CPLD [22]

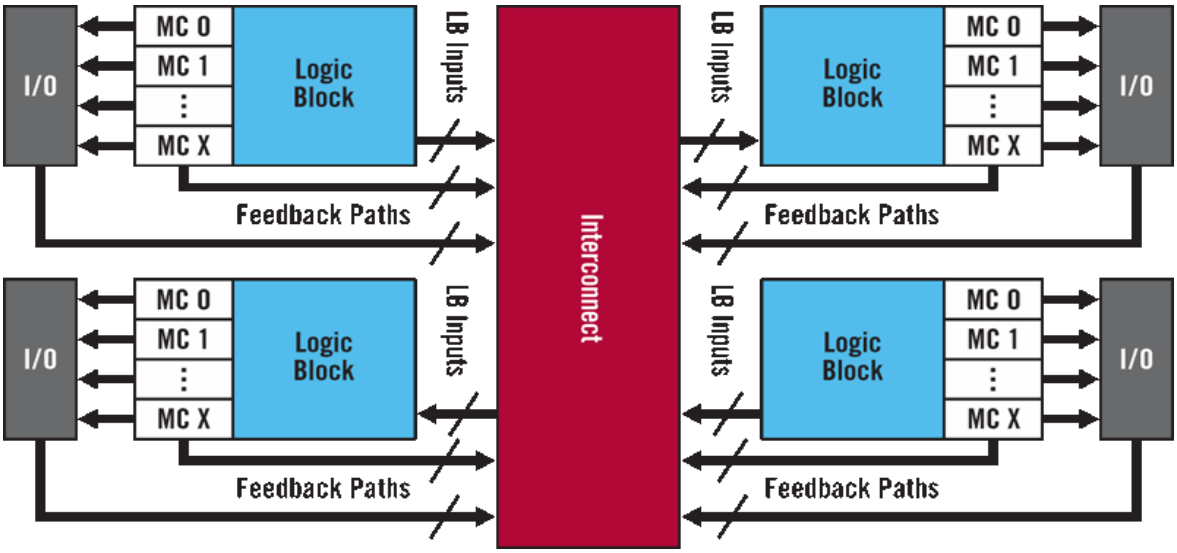


Figure 4-5 Internal Structure of a CPLD [23]

Field Programmable Gate Arrays

While PALs and GALs were being developed by various manufacturers, Ross Freeman, co-founder of Xillinx Technologies, was working developing of programmable hardware by using gate array technology. This led to the development of Field Programmable Gate Arrays (FPGAs) in 1985.

Internally FPGAs consist of three major blocks (a) Configurable Logic Blocks (CLBs), (b) I/O Blocks and (c) Programmable Interconnects. Along with these basic blocks modern FPGAs also consist of integrated memory (RAM and ROM), multipliers and DSP slices.

The CLBs, also known as Logic Elements (LEs), can be further broken down into (a) Look Up Table (LUT), and (b) a memory cell. The LUTs in the logical cell are used to implement the different sequential or logical functions such as a shift register or a memory access block. The memory cell on the other hand can be configured to act as an edge triggered flipflop or latch, so as to detect rising/falling edges or the level of a certain signal.

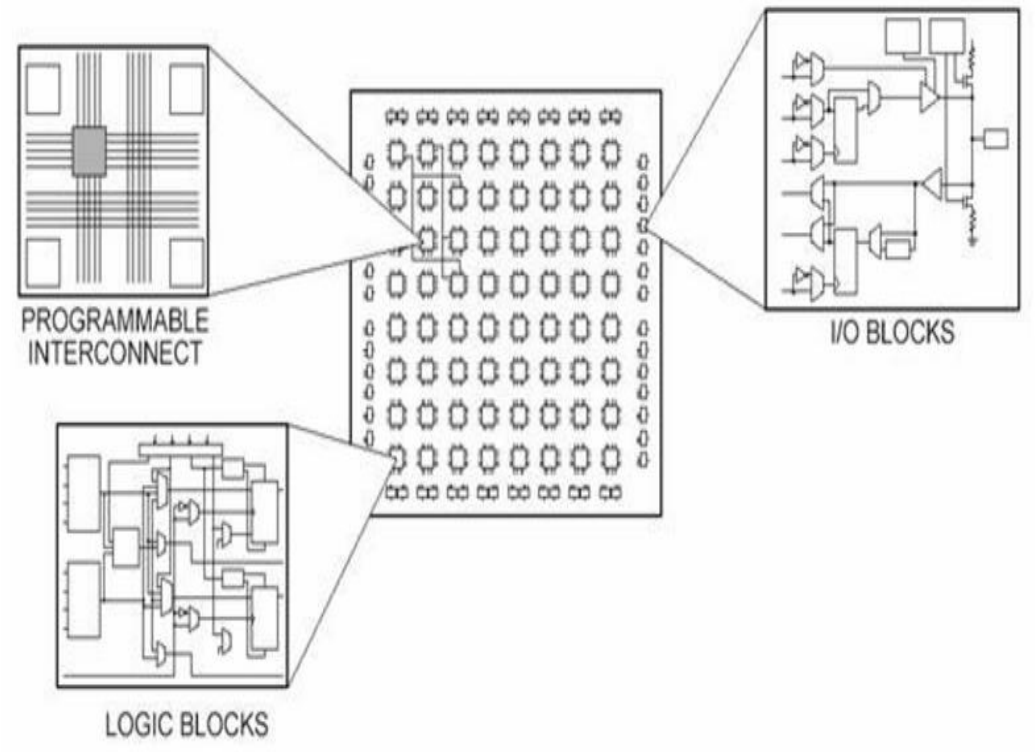


Figure 4-6 Building blocks of FPGA [24]

The I/O blocks allows the I/O pins to be programmed into a certain mode i.e input, output or even as a bi-directional pin. It also allows to implement different standards such as 3.3V LVTTL, 1.8V LVCMOS, PCI or standards such as LVDS and RVDS. Programmable I/O pins allows the design to be many times more flexible.

The programmable interconnects are what makes FPGAs to be as flexible. These interconnects allow the different blocks to connect to each other. Any CLB in the FPGA can be connected to any CLB via these interconnects. The programmable interconnects also connect the I/O blocks to the CLBs. Internal to the CLB, the connections are also programmable and are routed as required.

Internal to the CLBs, memory elements can be used to implement arrays to store data sets. However, due to the interconnect routing and individual memory blocks, this is

often very resource intensive. Thus, modern FPGA integrate memory blocks which, depending on the device, can be configured into different sizes from 16 bytes to 32KB.

Though multiplication may seem as a mundane task, it is highly resource intensive and complex to implement in hardware. Hence, to preserve CLBs and optimize performance, modern FPGAs consist of multiplier blocks, which allow multiplication of 8- to 64-bit numbers.

Advantages of Programmable Hardware

The top five benefits of using programmable hardware (esp. FPGAs), as described on the Xilinx webpage [24] are as follows:

1. Performance

   Programmable hardware allows true multi-tasking. While running a program on either a computer or on an embedded system, instructions are executed sequentially. But when such tasks are implemented using hardware parallelism, performance gains of more than ten times can be easily obtained.

2. Time to market

   With development on FPGAs, rapid prototyping and flexibility can be obtained. Ideas can be tested and implemented on FPGAs without developing custom ASIC via. the long fabrication process. Using modern software tools, IP cores and commercial off-the-shelf hardware, solutions can be implemented within weeks.

3. Cost

   Many of the custom hardware such as ASICs, need not be manufactured in bulk. The development of such hardware can cost from a few thousand dollars to millions of dollars depending on the complexity. Such hardware devices can be implemented on FPGAs which exclude any manufacturing costs.

4. Reliability

   Micro-processor based systems often involve many stages of abstraction, such as a RTOSs which run scheduling and task sharing functionality. It also includes drivers which control the various hardware resources and memory management unit can cause time critical to be stalled. Using FPGAs, all tasks can be performed in parallel, such that there is no interference or dependence making the design more reliable.

5. Long-term Maintenance

   FPGAs chips as mentioned in the above section are Field Programmable, this allows the functionality of the devices to be updated to keep up with new standards or customer requirements without much overhead costs. Also, since these devices can be programmed in the field, via Ethernet or USB, on-site support may not even be necessary.

Chapter 5

The Java Bytecode Execution Engine (JBEE)

Introduction to JBEE

The Java Virtual Machine during runtime converts the bytecode instructions into native processor instructions, which are then executed on the native processor. This translation performed by the interpreter is what makes high level languages such as Java machine independent. Along with the interpreting bytecodes, the JVM also performs tasks such as class loading, which is done on demand, bytecode verification and resource allocation, garbage collection and Just-In-Time (JIT) compilation. Since, all these functions are performed during runtime, Java programming does not provide optimal performance in terms of throughput (speed).

To overcome the performance issue with Java, a processor can be designed such that the Java bytecode is the native processing language for the machine. This would eliminate the need to translate the instructions. Such a processor could work independently executing directly java source files, or even be used as a co-processor. Such implementations of JVM are known as Java Processors.

Many operations such as memory allocation, garbage collection and object instantiation performed by the JVM, cannot be performed independent of an operating system. With a pure hardware implementation, such processes can be handled much faster, further increasing throughput.

The JBEE is one such implementation of a Java processor on an FPGA. The JBEE can be considered as a stack based micro-coded processor, which breaks down Java bytecodes into a sequence of micro-instructions. The JBEE preloads the class file into memory, so as no runtime memory allocation needs to performed, further improving throughput.

Similar Projects:

Java processors usually find applications in the field of embedded systems. Some of the popular Java processors are:

1. picoJava

picoJava [27] more than being a Java processor is a microprocessor specification for the native execution of Java Bytecode. It was first introduced by Sun Microsystems in 1997, later a revision picoJava-II was released in 1999. The aim of picoJava was to speed up the execution of bytecode compared to a standard CPU. Theoretically, a speed up of up to 20 times be been achieved.

A processor based on this specification, could also have executed C/C++, and the design was intended to be a Reduced Instruction Set Computer (RISC). The specification, however, did not include any I/O and memory requirements or standards.

An open source version of PicoJava was implemented on a FPGA, but is no longer available.

2. Cjip by IMSYS [28]

The Cjip was developed to provide the Java on chip functionality. It was released in 2000 by a Swedish company called IMSYS. The Cjip microprocessor was developed to support multiple instruction sets for languages such as Java, C, C++.

Some of the features of Cjip are it would allow native execution of Java Bytecode, garbage collection as a native firmware process and Virtual peripherals called Veripherals. The Cjip was a complete J2ME (Java 2 Micro Edition) solution. The processor also allowed interrupt driven tasks to run along with the Java application.

It is suited for internet applications, where memory, size and power consumption are constraints.

3. aJ100 by ajile Technologies [29]

The aJ100 was the first processor to implement both native processing of Java bytecode and Java multithreading on a single chip. The main purpose of development of the aJ100 was to allow developers to take advantage of the compactness of Java on a processor for embedded applications. The aJ100 provided a low-power 32-bit Java core, with integrated memory and peripherals.

The main advantage of using the aJ100 was it that could function without an RTOS which allowed a smaller memory footprint and much faster performance as compared to a direct Java application. With true hardware multitasking, thread switching could be obtained very fast, under 1ms. This allowed the processor to be used in applications which had hard real-time constraints. It also supported multiple JVM instances, to allow programs to be run simultaneously.

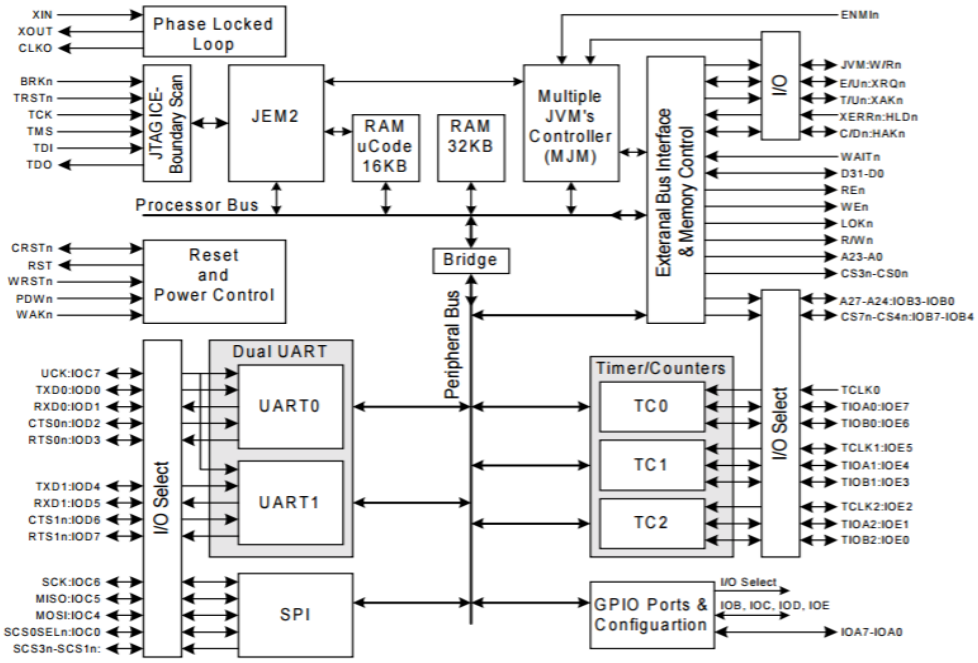


Figure 5-1 Internal architecture of aJ100[30]

The aJ100 was developed to support smart mobile devices, automotive applications and controllers in a network in an industrial environment.

4. Komodo [31]

The Komodo is a multi-threaded, pipelined (4-stages) Java processor. The Komodo system is based on middleware which allows JVM like features in addition to APIs and the microcontroller itself. The system could support direct execution of Java Bytecodes and allow multiplying the hardware for optimized performance.

The 4 stages of the pipeline were fetch, decode, execute and write-back, allowing a simple design and features such as priority management using a hardware scheduler, interrupt handling from help with the signal unit.

The Komodo system was developed with the intention of researching real-time scheduling on a multi-threaded processor.

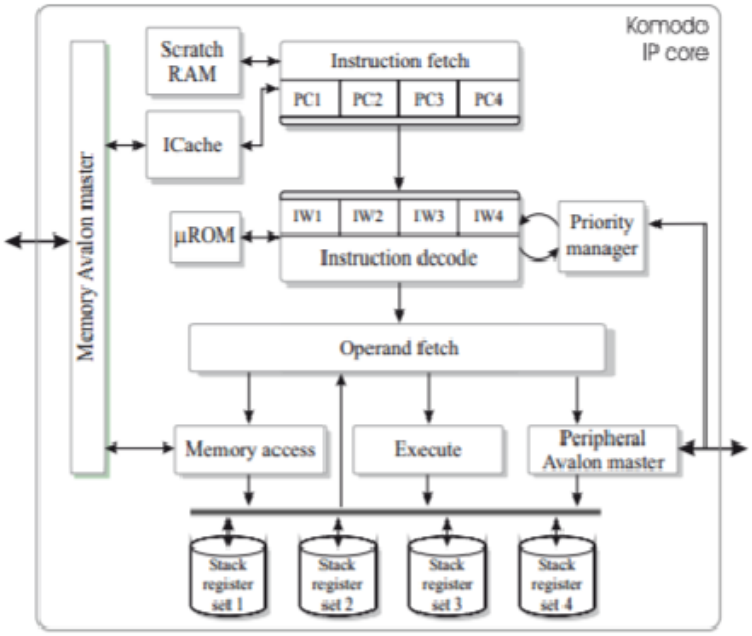


Figure 5-2 Komodo Java Processor Core [32]

Limitations of the JBEE

The Java Programming Language was developed with the intent of a programming language for embedded devices, which ran with the support of an embedded Operating System(OS). The idea never went forward and later Java was released as a programming language for the Internet. However, it was always intended to run in conjunction with an OS.

The Java Bytecode has certain instructions such as `invokestatic`, `invokeinterface` and `anewarray` invoke and or create and initialize a new object, are too complex to be implemented in hardware alone. Hence, no hardware implementation of JVM exist yet, which can implement all the bytecodes.

Building an JVM, without an OS, direct access to memory and I/O's along with a specific hardware architecture is needed. For low level access i.e. memory and I/O, Java uses native functions, which are functions written in a language native to the processor such as C and C++.

The JBEE, being a pure hardware implementation and hence, ignores such bytecodes.

Java, being an object-oriented language, is usually used in programs with multiple classes, interfaces and field variables. But as mentioned above, without an underlying OS these cannot be implemented. Hence, the JBEE is only designed to handle programs with single class, with no support for interfaces and field variables.

Furthermore, to avoid complicated FPGA implementation, support for floating point numbers, both float and double datatypes, has been omitted.

Chapter 6

Class File Processing

In the JBEE, to allow for faster run-time by elimination of runtime dependencies, the class file is pre-processed. This includes eliminating run-time dependencies, omitting metadata and converting the file into proper format. In this chapter, we look at how this is done for this project.

Bytecode Extractor

Since, the JBEE is a limited implementation of the JVM, the Java class file cannot be directly loaded into the processor. The class file has to be broken down into separate files containing the constant pool and methods.

The Bytecode Extractor is a Python program which breaks down the class file into the constant pool, fields, interfaces and the different methods. As, the JBEE only deals with the constant pool and methods, only these are written to bytecode files. The remaining class file data is ignored.

As the class file consist of a stream of 8-bit values, the program reads the required number of consecutive bytes to construct the 8-bit, 16-bit and 64-bit data values.

The Bytecode Extractor Program

The Bytecode Extractor program first checks the validity of the class file by reading the magic, a 4-byte quantity, which equals the value 0xCAFEBABE. If the first 4-bytes match the magic number, the file is considered to be a valid class file.

The next 4-bytes give the version of the class file. The version is split into two 2-byte values the minor and the major version numbers. These values indicate the version of the Java compiler.

The next section of the class file is the constant pool. A separate class is used to extract the elements of the constant pool. The first two bytes indicate the number of

elements of the constant pool. This value is then used as the count for iterating through the constant pool.

The first byte of each entry of the constant pool is the tag. The value of the tag indicates the type of element. The following table indicates the tag value and the associated type:

Constant Type	**Value**
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

Table 6-1 List of Constant type and corresponding values [15]

Each tag byte is followed by two or more bytes which provide additional information about the specific entry such as the name, length of the entry etc. The format of this additional information is indicated by the tag. In the case of a integer, long integer, float or a double entry the size of the entry is fixed, but in the case of the strings the next two bytes determine the size of the string.

The JBEE only supports datatypes are the integer and long integer, these are added into the constant pool bytecode file (const_pool.byc). The integer constants are 32-bit values stored at a single index. The long elements are split up into two indices, and follows the Java convention of big-endianness. The index of the element in the constant pool along with the byte are added to the file serially as they appear in the constant pool. The remaining entries of the constant pool are ignored.

After the constant pool, the next two bytes of the class file is the access_flags entry, which as the name indicates indicate the access flags for the constant pool. These access flags are masked and need to be extracted from the two bytes. The following table indicates the mask and the associated flags:

Flag Name	**Value**	**Interpretation**
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared final; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the invokespecial instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared abstract; must not be instantiated.

Flag Name	Value	Interpretation
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an enum type.

Table 6-2 List of Flags and corresponding mask value [15]

Following the access flags, the next two bytes are a pointer into the constant pool of the same file which point to a `CONSTANT_Utf8` entry, specifying the name of the class file. This entry is called `this_class`.

Similar to the `this_class` field, the next field is the `super_class`. This field also points to an entry in the constant pool to a `CONSTANT_Utf8` entry, specifying the name of the super class.

The subsequent entry in the class file is the `interface_count` and the `interfaces[]` array. These field indicates the interfaces that are direct superinterfaces to the current class or interface. The `interface_count` is a two byte entry which gives the depth of the `interface[]` array. Each entry in the `interface[]` array is two byte wide and points to a `CONSTANT_Class_info` entry in the constant pool.

Just as the `interface_count` and the `interfaces[]` array, the `field_count` and the `fields_info[]` structure indicate the fields in the class file. These fields include both class variables and instance variables. The `fields_count` gives the number of `field_info` structures. The `field_info` structure has the following format [15]:

```
field_info {
    2-bytes        access_flags;
    2-bytes        name_index;
    2-bytes        descriptor_index;
    2-bytes        attributes_count;
```

```
        attribute_info attributes[attributes_count];
    }
```

The access_flags are similar to the access flags field for the corresponding field. The name_index and the descriptor_index fields points to an entry in the constant pool of the type CONSTANT_Utf8_info, indicating the name of the field and a valid field descriptor respectively.

The attributes_count indicates the number of additional attributes associated with the field and the attributes_info structure gives the specific type of attribute. The attributes defined by this specification could be a ConstantValue, Synthetic, Signature, Deprecated, RuntimeVisibleAnnotations and RuntimeInvisibleAnnotations.

Though the above mentioned fields are not directly used in the JBEE, the Fields class in the program isolate the fields and other sections of the class file but are not written to any file. If a future modification of the JBEE supports fields, interfaces and classes, these can be easily obtained from the program and written to a file.

Subsequent to the fields, is the method section of the class file. Like the fields, methods are obtained by a two-bit method_count and a method_info[] structure. To extract methods out of the class file, a method_info class is used. This class reads the method_count from the class file and loops until all the methods are extracted. The method_info structure has the following format [15]:

```
    method_info {
        2-bytes         access_flags;
        2-bytes         name_index;
        2-bytes         descriptor_index;
        2-bytes         attributes_count;
        attribute_info attributes[attributes_count]; }
```

If the attribute of the method is ‘CODE’, it indicates that the following bytes are the bytecodes the JVM execute. These are bytes are added to the respective method files. The files are named sequentially as ‘method_<value>.byc’ where the value is an integer number. Each bytecode file contains the name of the method and code line. The code line contains all the bytecodes associated to the method.

The final field of the class file is the attributes section, and follow the same format as the methods and fields. These attributes are used by JVM implementation to obtain additional details about the class file. And thus, are ignored by the bytecode extractor file.

Memory Initialization File and Generator

The Memory Initialization File

To initialize the contents of the method_area_ROM and the constant_pool_ROM, the Altera Quartus II IDE supports two file types (a).mif file and (b).hex file.

For the implementation of the JBEE, the .mif file was selected.

MIF stands for Memory Initialization File. It is an ASCII text file, used to specify the initial contents of a memory such as RAM, ROM or CAM, with the .mif extension. The memory initialization file contains the contents of each address possible in the memory. Before specifying the contents of the memory, must specify certain parameters about it. These parameters include the memory depth and the memory width, and the radix used to define these values throughout the file. The radixes may be binary, octal, hexadecimal.

The Quartus II IDE uses the Memory Initialization File as an input during the simulation and the compilation process. A separate file is required to initialize each block of memory.

The following table shows the different formats that allow data and address combinations:

Address : Data Pairs Syntax Rules	Definition	Example
A : D	Addr[A] = D	2 : 4 Address: 01234567 Data: 00400000
[A0..A1] : D	Addr[A0] to [A1] contain data D	[0..7] : 6 Address: 01234567 Data: 66666666
[A0..A1] : D0 D1	Addr[A0] = D0, Addr[A0+1] = D1, Add [A0+2] = D0, Addr[A0+3] = D1, until A0+n = A1	[0..7] : 5 6 Address: 01234567 Data: 56565656
A : D0 D1 D2	Addr[A] = D0, Addr[A+1] = D1, Addr[A+2] = D2	2 : 4 5 6 Address: 01234567 Data: 00456000

Table 6-3 Table representing different data input formats for.mif files [33]

The Memory Initialization File Generator Program

To be able to load files in the FPGA processor, a .mif file needs to be created from the bytecode files generated by the Bytecode Extractor. This conversion is performed by the. mif file generator, a Python program. The file generator uses the hexadecimal as the default radix, and specifies the content (data) to be loaded into each memory location individually. The unused memory locations are loaded with a zero.

Chapter 7

Architectural Overview of the JBEE

The JBEE is an implementation of the JVM on an FPGA. It is a single core, stack machine with integrated memory. The important features of the JBEE are:

- Integrated RAM and ROM memories
- Pseudo 2-stage pipeline
- Multi-cycle instruction processing
- State machine based implementation
- Low I/O requirement (3 pins)
- Pre-Decrement/Post-Increment Stack

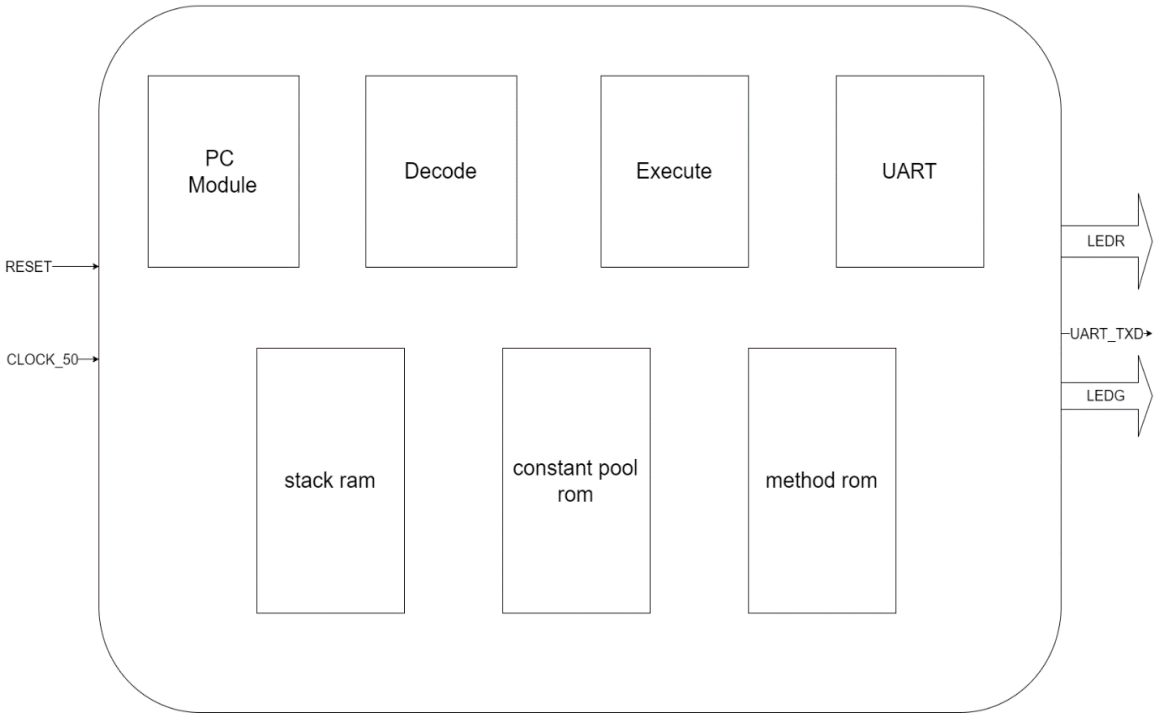


Figure 7-1 Overview of the JBEE

Components of the JBEE

To simplify the process of designing the JBEE and to be able to understand the workings better, the JBEE has been broken down into important functional modules. These modules are:

- PC module
- Memory Area
- Decode Module
- Execute Module

Memory Areas

The memory from which the JVM retrieves the class file or to which it writes the results and0 objects is known as the run time data area. This data area for the JVM is divided into 5 areas, (a) the method area; (b) heap (c) Java Stack; (d) Program Counter registers; (e) native method stack.

The method area is the memory into which the methods and the related contents such as the constant pool, access flags, fields, interfaces etc. are loaded. The stack stores the stack frame and the local variables for the respective class. The heap consists of the fields and the objects created during runtime. The program counter is a register which points to the location in the method area from where the next instruction is to be read, while the native method stack is used in case of function calls to a native language (such as C, C++) function are made.

To maximize effectiveness of the FPGA, the data areas in the JBEE had to be reorganized. The JBEE areas are, (a) PC (b) Method ROM, (c) Constant Pool ROM, (d) Stack RAM.

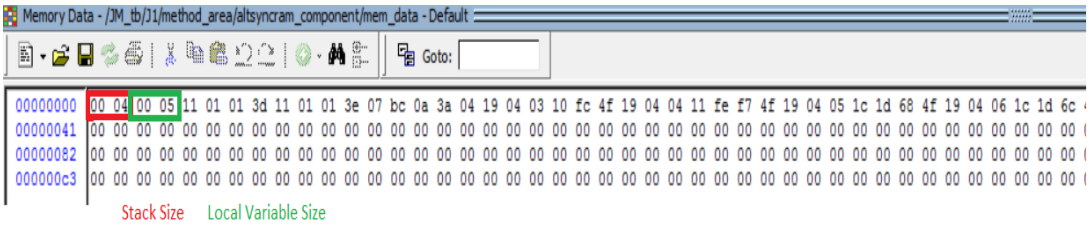
PC

The PC in the JBEE is a simple 16-bit register which points to the location in the method area, indicating from where to fetch the next byte.

Method ROM

The method area, in the JBEE is an 8-bit wide memory. The default size of the memory is 256 words, which can be extended to be a much larger memory. Since, the memory is 256 words deep and 8 bits wide, the address bus and the data bus are both 8 lines. The word size is chosen to be 8 bits as the bytecode instructions defined by Java are 8 bits, as well as is the additional data (such as index, constant).

The Method load is loaded from a file named method.mif. The file contains the depth of the stack needed for execution of the bytecode, the size of the local variable array and the bytecode associated with the method. The first two bytes of the ROM indicate the stack depth and are loaded into the stack_size register, the next two bytes give the size of the local variable array which is loaded into the LP register, what follows is the bytecode for the method.



(a)

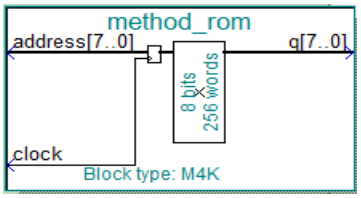


Figure 7-2(a) Sample data for method_ROM (b) Input/Output buses for method ROM [34]

The register/buses associated with the method ROM are:

`method_area_bus`: The `method_area_bus` is an 8-bit bus which acts as the address bus for the method ROM.

`method_area_out`: The `method_area_out` is an 8-bit bus which acts as the data bus for the method ROM.

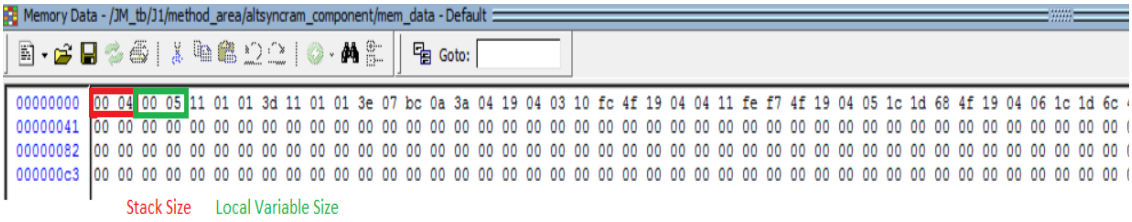
PC

The method_area_bus of the method ROM is driven by the PC. Traditionally, the PC only points to instruction that need to be executed, but in case of the JBEE, an offset is present. This is handled by using the current_PC register in the execute module.

Constant Pool ROM

The Constant Pool in the JVM is part of the method area. To improve performance and reduce memory read latencies, the constant pool in the JBEE is loaded into a separate ROM. The constant pool ROM by default is 64 words deep, and each word in the ROM is 32-bits wide. The width of the constant pool ROM is set to be 32-bits as the constant pool in the class file is organized as 32-bits. A byte, short or an integer stored in the memory occupies a single memory location, while a long occupies 2 locations. Also, the 32-bit words reduces the number of reads to the memory i.e. to read an int from the ROM, it only takes a single clock cycle, whereas if the ROM were organized as 8-bit words, it would have taken a minimum of 4 clock cycles.

The constant pool ROM is loaded from a file named the constant_pool.mif. The constant pool ROM only store integers and longs, whereas the constant pool in the class file also contains references, strings and fields. The elements in the constant pool are accesses by their indices, and thus the location cannot be altered. Hence, the constant pool ROM contains many of empty slots.



(a)

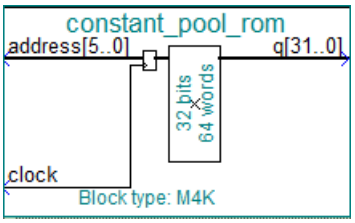


Figure 7-3 (a) Sample data for Constant Pool ROM (b) Input/Output buses for Constant Pool ROM [34]

The register associated with the constant pool ROM are:

constant_pool_bus: The constant_pool_bus is a 6-bit bus which acts as the address bus for the constant pool ROM.

pool_pointer: The pool_pointer is a 6-bit register which drives the address bus for the memory. The register is part of the execute module.

constant_pool_out: The constant_pool_out is a 32-bit bus which acts as the output data bus for the ROM.

Stack RAM

The Stack area in the JVM contains only the method stack frame and the local variables array. To reduce the complexity of the JBEE and to reduce the number of LEs used, the heap has been merged along with the stack frame and the local variables array. The stack RAM is a 32-bit wide, just like the constant pool ROM, and has a depth of 64 words. The 32-bit width allows a byte, short or an integer to be stored in a single location,

while a long needs 64 bits. Since, the stack RAM is not involved in any compile time updates, it does not need to be initialized, and is empty at the start of the execution.

The heap in the JVM, is used to store objects (class instances) and arrays. As the heap size cannot be predetermined, the memory allocated to the objects and arrays which are not in use any more needs to be freed, to limit the runtime memory used by the program. This is done by the garbage collector. Since, the JBEE does not support objects and is independent of an operating system, the garbage collection is not implemented.

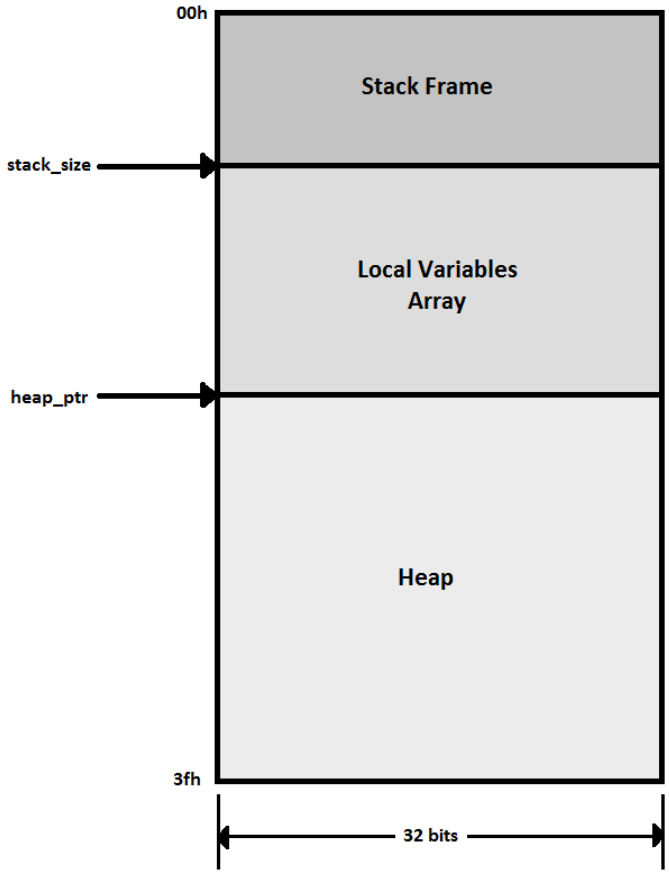
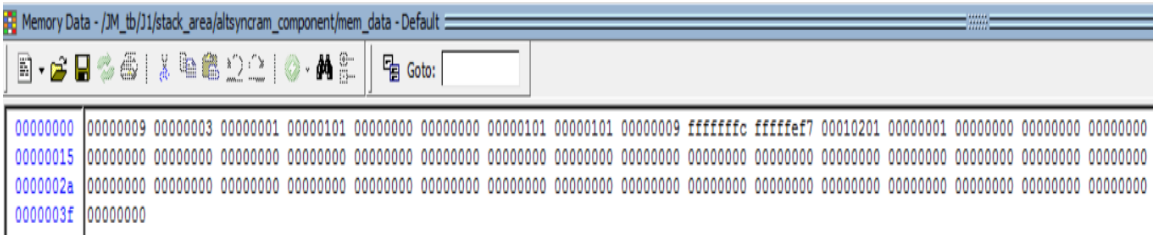


Figure 7-4 Organization of stack memory



(a)

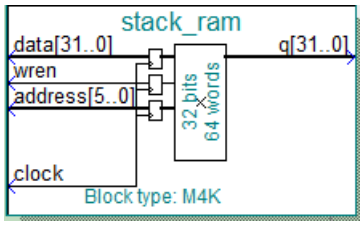


Figure 7-5 (a) Sample data for Stack RAM (b) Input/Output buses for Stack RAM [34]

The size of the stack frame and the local variable array is predetermined during compile time. Hence, the stack RAM can be organized to the sizes specified in the method ROM. The remaining of the stack RAM can be allocated to the heap. Since, there is no garbage collector the heap may run out of bounds, and proper care must be taken by the programmer to avoid such erroneous conditions.

The registers/buses associated with the Stack RAM are:

stack_bus**:** The stack_bus is a 6-bit bus which acts as the address bus for the Stack RAM.

stack_bus_select: The stack_bus_select is a 2-bit input to the stack bus arbitrator, which choses from one of the three inputs (a) stack_adder_out, (b) local_adder_out or (c) heap_adder_out. The stack_bus_select is driven by the state machine and depending on the state the value of the bus changes.

Stack_data_in_reg and Stack_data_in: The stack_data_in is a 32-bit bus which acts as the input data bus for the Stack RAM. The stack_data_in bus is driven by the stack_data_in_register.

Stack_data_out: The stack_data_out is a 32-bit bus which acts as the output data bus of the Stack RAM.

stack_write: The stack_write acts as the write enable for the stack RAM. The stack_write is an active high signal which means when the stack_write is one the data on the stack_data_in bus is written to the address pointed to by the stack_bus.

PC module

The PC module contains the program counter, which points to the next location of in the method ROM from which a bytecode instruction is read. Depending on which location the PC points, the data is loaded either into the Instruction Register, Stack Size Register or the Locals Pointer (local variable pointer).

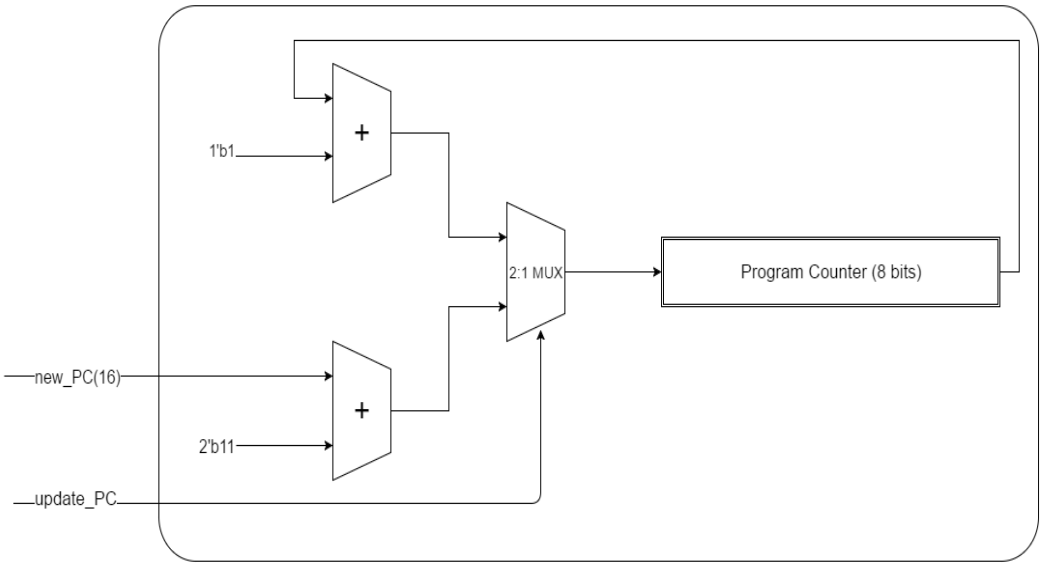


Figure 7-6 Overview of PC module

Architecturally, the PC, being a register, cannot be connected to the address bus of the method ROM directly. Hence, it is used to drive the net(bus) method_area_bus which acts as the address bus for the method ROM.

The fig. 9-6 shows the internal structure of the PC module. The module contains of the two adders, the increment_adder increments the PC by 1, whenever a byte is read from the method ROM is read, while the branch_adder updates the PC in case of a jump or a branch. The branch_adder adds 3 to the new_PC, which is generated by the execute module. The 3 is added to the new_PC, to compensate for the initial bytes of the method ROM indicate the stack size and the local variable size.

The PC_arbitrator_mux assigns the output of one of the adders to the PC register. The mux outputs are controlled by the update_PC signal.

Decode Module

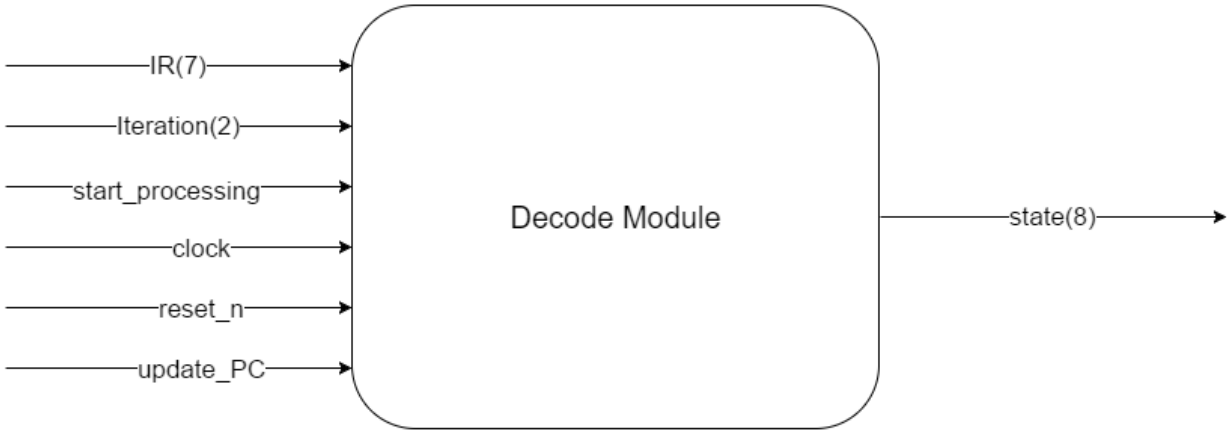


Figure 7-7 Input output representation of Decode module

The decode module, as the name indicates, is responsible for decoding the instruction and implementing the state machine.

Depending on the current state of the state machine, the current instruction (i.e. contents of the IR) and the iteration variable, the next state is calculated. In case of certain instructions, certain states of the state machine need to be repeated. In such cases, the iteration variable is used to keep track of the number of times a certain state or block of states has been executed.

The state machine begins when the start_processing flag get activated, after the stack_size and local_pointer registers are written into. The initial state loaded into the state machine is fetch and then the state is updated on the rising edge of the clock.

States of the State Machine

The state is an 8-bit register, allowing up to 256 states, out of which only 23 are used. The table below names the states and their equivalent decimal values.

STATE	**VALUE (decimal)**
fetch	0
write_stack	1
inc_SP	2
dec_SP	3
read_stack	4
write_local	5
read_local	6
byte_two_state	7
byte_three_state	8
byte_four_state	9
compare	10
update_PC_state	11
ALU_op	12
ALU2_op	13
inc_HP	14
write_heap	15

read_heap	16
read_pool	17
handle_long	18
return_state	20
exception	39
nop	254
done_state	255

Table 7-1 State and corresponding value of state register

fetch

The initial state during which the instruction is loaded into the IR is the fetch. During fetch, the bytecode which is present on the Method ROM data bus is written to the register.

write_stack

As the name indicates, in the write stack state the data is written into the stack memory. To enable the stack to be written to, the stack write enable signal, stack_write, is first set and the stack bus arbitrator is fed 2'b00, this results in the stack adders output to be fed to the stack RAM as the address, i.e. it connects the stack_add_out to the stack_bus. These two signals set up the stack to be written, and the data to be written to the stack has different sources depending on the current instruction being executed. This data is then written into the register stack_data_in_reg which is connected to the data bus of the stack RAM. On the next rising clock edge of the clock cycle, this data is written to the top of the stack.

inc_SP

The write stack state is usually followed by the increment SP state. The increment SP state deactivates the stack write, by resetting the stack_write signal. It then toggles the stack_add_enable signal to activate the stack adder. To signal the addition operation to

the adder/subtractor unit the stack_add_sub signal is set. The adder/subtractor increments the value of the stack by a 1, and this value is then written into the Stack Pointer (SP) register.

dec_SP

Similar to the increment SP stage the decrement SP stage decrements the value of the stack pointer by 1. It usually precedes the read stack stage. To enable the subtractor, the stack_add_enable signal is toggled and the stack_add_sub is reset. The adder/subtractor then decrements the value of the stack by a 1, and this value is then written into the Stack Pointer (SP) register.

read_stack

The read stack stage is where the contents of the stack, pointed to by the stack adder output. The data read by default is transferred to the stack_read_reg, unless specified to a different register for a certain instruction.

write_local

As in the JVM, the locals on the JBEE are stored in the stack area. Hence, to write to a local variable, the stack write enable is set. The position of the local variable in the Stack RAM is pointed by the Local Adder. Depending on the instruction the offset is either a constant or the value read from a register. This value is written to the local_offset register. To connect the local adder output to the stack RAM address bus, the bus arbitrator mux is fed 2'b01. Similar to the stack write state, depending on the instruction, the source of the data may vary. This data is written into the stack_data_in_reg. On the following positive edge of the clock cycle the data is written into the local variable.

byte_two, byte_three and byte_four

The Java bytecode instruction are variable length instructions, which may occupy from 1 to 4 bytes in the class file. For certain instructions such as wide, tableswitch and

jsr_w. the length of the instruction is not fixed, and thus cannot be implemented without an software assistance.

The byte_two, byte_three and byte_four stages are used to read the additional bytes of the bytecode instruction into the respective state registers, to make them available for further processing whenever needed. The data read from the method ROM is usually returned into byte_two, byte_three or the byte_four registers depending on the state, or into a different register if specified for the specific register.

read_local

As in the case of the write local state, to read the local variables, the address is fed by the local adder. Depending on the instruction, the offset is either a constant or the value read from a register. This value is written to the local_offset register. To connect the address bus to the output of the local adder, the bus arbitrator mux is fed 2'b01. The stack write enable is reset, to make sure no data is written into the stack unnecessarily.

Note: Unlike the stack read state, the value read is not directly written to a register. This is because the default pointer to the stack RAM is the stack_adder_out and in case of the read local, the data from the location is available on the next falling edge of the clock.

compare

The compare state is invoked only in case of a conditional branch instruction. The state is used to determine whether the branch should be taken or not.

Depending on the instruction, one of the six outputs of the comparator is checked to be true or false. If the condition is true, the update_PC signal is set, signaling the PC module that a branch needs to be taken and the PC needs to be updated. The new address to which the program counter jumps is calculated from the byte_two and the byte_three registers and stored in the PC_offset register. If the conditions fails, no action is taken, and thus, the PC will be incremented by 1'b1 at the end of the instruction.

ALU_op, ALU2_op

The ALU_op and the ALU2_op are states used to indicate to the ALU, which operation needs to be performed. The JBEE consists of two ALU, one for integer/long operations and the other for bitwise/logical operations.

When these states are invoked, depending on the instruction the ALU_op or the ALU2_op bus are set indicating to the ALUs which output should be routed to the result. The following table shows the operation to be performed and the value of the ALU operation bits:

update_PC_state

The update PC state is a special state used only in case of the GOTO instruction. It is used to calculate the new address of the PC, from the byte_two and the byte_three registers. It also sets the update_PC signal indicating the PC module that a branch has been encountered and the PC address needs to be updated.

inc_HP

The increment HP state is a special state invoked when an array is to be created on the stack. The only instruction it is valid for is the newarray instruction. For the newarray instruction, the second byte of the instruction (loaded in the byte_two register) determines the datatype of the array. If the datatype is either byte, short or integer, the heap pointer needs to be incremented by only the size of the array, which is loaded on the stack. For the long datatype, the HP must be incremented by twice the size of the array.

write_heap

The write heap state, is used to write to an array element. The position of the array element on the heap (in the Stack RAM) is determined, by the values pushed on the stack. The location of the array element is pointed to by the heap adder. The bus arbitrator mux is fed 2'b10 to connect the stack RAM, to the output of the heap adder. To enable the stack

RAM to be written to the stack_write signal is set. Similar to the write local and write stack operation, the data is written onto the RAM on the following rising edge of the clock.

read_heap

The implementation of the read heap state is just like that of the read local. The stack_write is first deactivated, to avoid unintentional writes to the memory. To read from the array element, the output of the heap adder must be connected to the address bus of the stack RAM. This is done by feeding the bus arbitrator mux by 2'b01.

Note: Similar to the read local stage, the data read is not written to any register, this is handled during the stage that follows the read heap.

read_pool

The constant pool in the JBEE is a separate memory area, unlike the JVM. This makes reading the data out of the constant pool considerably quicker. During this stage, the pool_pointer register is set to the location of the element in the constant pool ROM. This address is calculated by using the byte_two and byte_three registers. The pool_pointer is connected to the address bus of the constant pool ROM. The data to be read is then available on the data bus on the next falling edge.

return

The return state is a special state that indicates the execution has ended. During this stage the return signal is set, indicating to the PC module to stop incrementing the program counter.

exception

The exception state is not invoked in case of regular operation of the state machine. In case an unsupported instruction or an error condition occurrence, the exception state is reached. The JBEE stays in the exception state until reset.

nop

nop stands for No operation. As the name indicates, no operation is performed during this state. This state is usually invoked by the state machine, when the processor is waiting for the data to be made available on the data bus of the method ROM.

handle_long

Handle long is a special case for handling the long datatype. The state is valid in only two instructions, ldc2_w (load long from constant pool) and lastore (store long into an array). If the instruction is ldc2_w the state increments the constant pool_pointer between the two reads. But for the lastore, the array_index is incremented between two writes to the heap.

done

The final state of each instruction is the done state. This state indicates that the processor is ready to fetch the next instruction. During this state, the instruction is committed i.e. if the instruction was a branch, depending on weather the branch is taken or not, the PC is updated in this state. Else, it updates the current instruction register, to keep track of instructions executed.

Execute Module

The execute module has been broken down into sub-modules to facilitate both design and implementation. The modular approach also allows reusability, which reduces the LE usage in the FPGA, which can allow implementing additional features. The different submodules are shown in figure 7-8 and explained below.

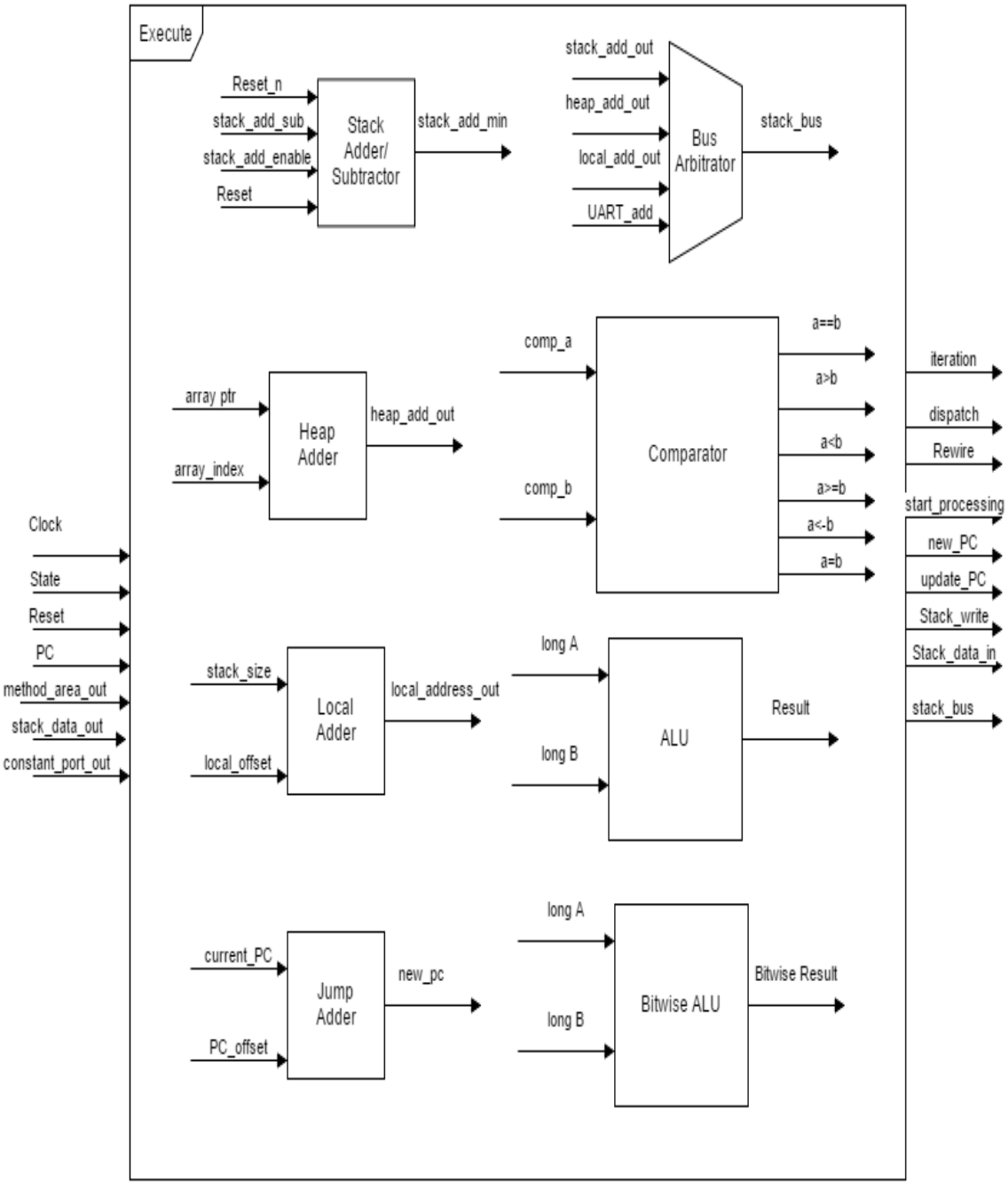


Figure 7-8 Overview of execute module

Stack RAM access sub-system

As described in the memory section, the stack consists of not only the stack frame and the local variables array, but also the heap. The accesses to each of these must be independent of the other. If the system was to use only one pointer to the Stack RAM, that would result in catastrophic failure conditions. As an example, consider a stack read followed by the local variable write. Initially, the pointer points to the top of stack to read the data from the RAM and the data is then stored in an intermediate register. This register is then updated to point to a local variable, causing the top of stack to be lost. Hence, access to one section is kept independent of the other. The size of the stack RAM is loaded into the stack_size register and that of the local variables is loaded into the local_size register.

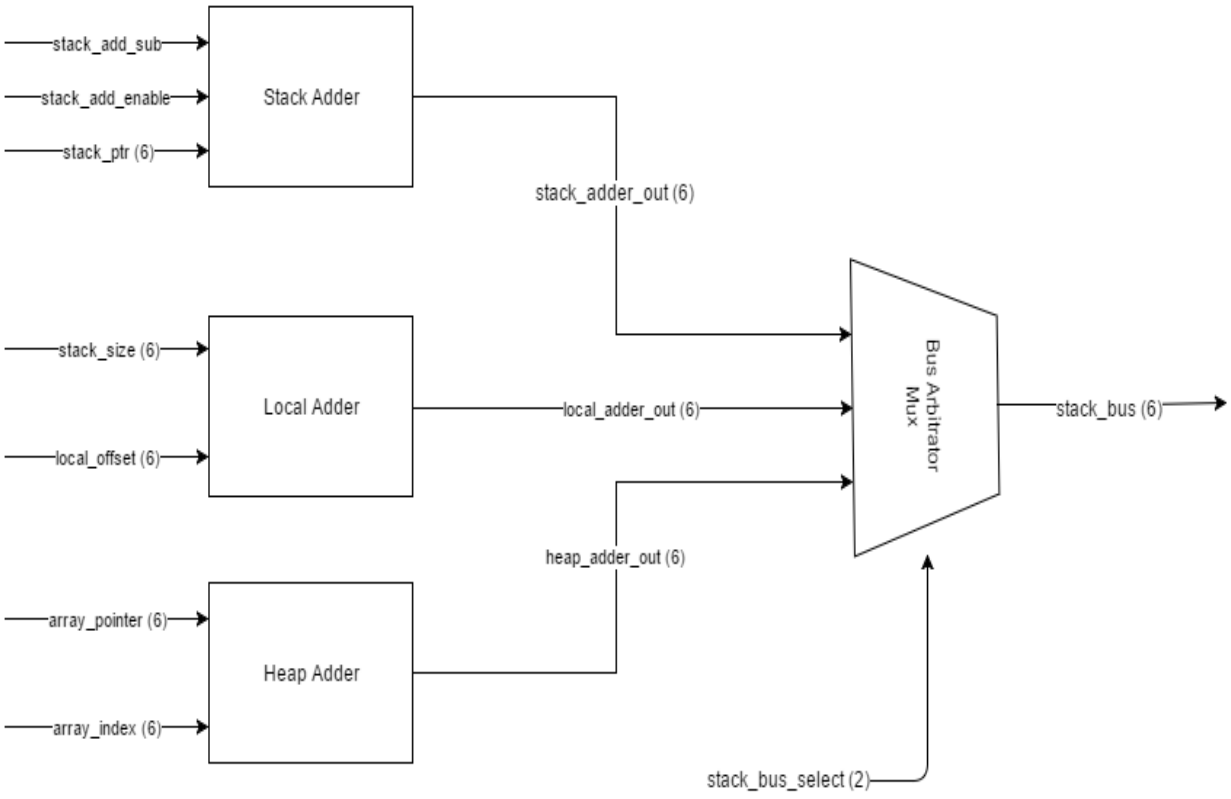


Figure 7-9 Overview of stack access subsystem

Stack Adder/Subtactor

The stack adder is used to point to the top of stack during a stack read or write. The output of the stack adder, stack_adder_out, is connected to the first input of the bus arbitrator. The inputs to the adder are (a) stack_add_sub signal, (b) stack_add_enable, (c) stack_ptr. The stack_add_sub bit enables the increment (by 1'b1) operation when set, otherwise when reset leads to a decrement (by 1'b1) operation. A change in state on the stack_add_enable signal triggers the stack adder operation. The stack pointer stores the current top of stack. The stack_adder_out depending on the last stack operation either leads or follows the stack_ptr.

Stack Operation

The stack in the JBEE, has been designed as a pre-decrement and post-increment configuration. Thus, before a valid stack read can be made the stack pointer must be decremented, to read from the top of the stack. After a valid stack write, the stack pointer should be incremented to point to the next location.

Conventional stack memory is located at the end of the memory and as the stack grows upwards, so as to avoid overlaps between the heap and the stack. Unlike the conventional stack, the JBEE stack begins at the start of the memory and grows downwards. This is possible since the maximum depth of the stack is predefined in the class file.

Local Adder

The local adder, during an instruction, is used to point to the location in the stack RAM from which data needs to be written to or read from. The output of the local adder output local_add_out is connected to the second input of the bus arbitrator mux. The two inputs to the local adder are the stack_size register and the local_offset register which is

updated during either the read or write local state. The stack_size register, once loaded, does not change for the respective method and can be considered as a constant.

Local Variable Operation

The local variables array is the second section of the stack RAM. The size of the local variables array is stored in the local_size variable. The local variables array begins at the location where the lowest stack top could exist. This location is pointed to by the stack_size register. The stack_size variable is loaded before the program start executing. The Java bytecode specifies the exact element in the which the data must be loaded. This acts as the offset from the start of the stack_size variable, indicating the exact location in the stack RAM to which the data needs to be written.

Heap Adder

The heap adder is used to point to a location in the heap to write or read an array element. The Java bytecode instruction load stack frame with the array index and a reference to the start of the array whenever an array element needs to be accessed. This allows the JBEE to read the reference and load it into the array_ptr register and the index and to load it into the array_index register. The array_ptr and the array_index registers act as the input to the heap adder. The output of the heap adder, heap_adder_out is connected to the third input of the bus arbitrator mux. The heap adder and the local adder are similar, as they generate an output when the appropriate state is reached. But the heap_ptr may not be a static or OTP register like the stack_size register and may be updated when more than one arrays are created.

Heap Operation

The heap pointer is a register which points to the location in the Stack RAM, from which a new array should begin in the case of an array declaration. The heap is allotted at the bottom of the Stack RAM, to allow the heap to grow dynamically during runtime. The

heap pointer is, thus, initialized to a value equivalent to the sum of the stack size and the local pointer. During runtime, whenever a newarray instruction is encountered, the heap pointer is incremented by the size of the array. The byte_two of the instruction gives the data type of the array, and the size of the array is loaded onto the stack. If the datatype is either byte, short or integer, the heap pointer needs to be incremented by only the size of the array, which is loaded on the stack. For the long datatype, the HP must be incremented by twice the size of the array.

Stack Bus Arbitrator

The stack bus arbitrator is a 3 input 6-bit mux with a single 6-bit output. The three inputs to the mux are, as mentioned above, (a) stack_adder_out, (b) local_adder_out and the (c) heap_adder_out. The select lines to the mux are named stack_bus_select, and has a width of two bytes, as there are three inputs to the mux. The state machine drives the stack_bus_select, and depending on the state the one of the inputs is latched to the output. The output of the stack bus arbitrator is connected to the address bus of the stack RAM.

Comparator sub-system

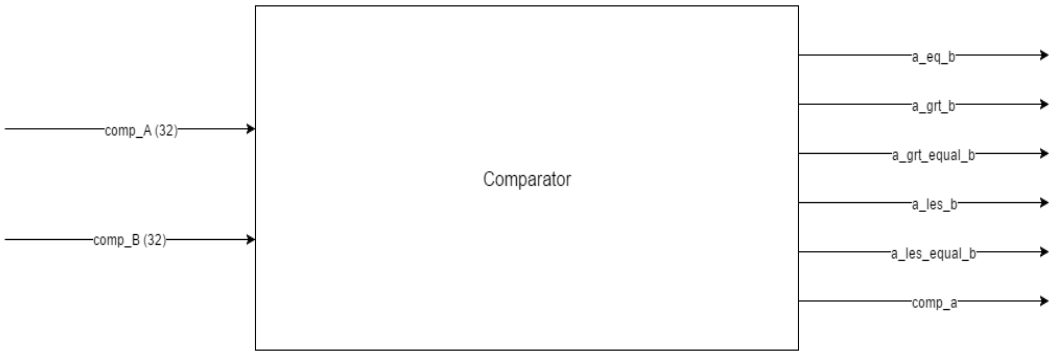


Figure 7-10 Overview of comparator submodule

The comparator in the JBEE compares two 32 bit values and generates six outputs (a) a_eq_b, (b) a_grt_b, (c) a_grt_eq_b, (d) a_les_b, (e) a_les_eq_b and (f) a_nt_eq_b. All the outputs are generated simultaneously in the comparator. The comparator has a

dedicated state in the state machine, where the needed output is checked, and depending on the value, further action is taken. The comparator plays a major role in integer/long compare instructions, compare with zero instructions and in conditional jump instructions.

Arithmetic Logical Unit

The JBEE uses an unconventional design for implementation of the ALU. The ALU is split into two parts (a) the arithmetic unit and the logical unit.

Arithmetic Unit

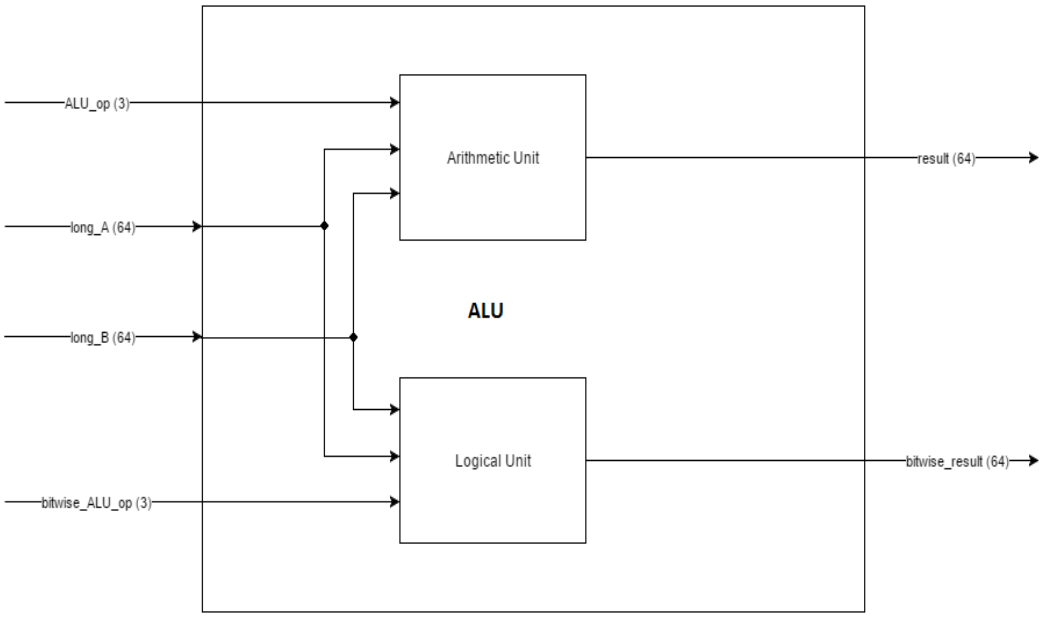


Figure 7-11 Overview of ALU submodule

The Arithmetic Unit in the JBEE performs (a) addition, (b) subtraction, (c) multiplication, (d) division and (e) modulo. The operands are fed to the arithmetic unit through long_A and long_B, which are 64-bit registers. As the registers are 64-bit wide, even long arithmetic can be performed in a single cycle. For integer arithmetic, however, proper care must be taken to sign extend the values when they are loaded into the registers. Similar sign extensions are performed in case of a byte or a short data type operation. The output of the arithmetic unit is available on a 64-bit result bus.

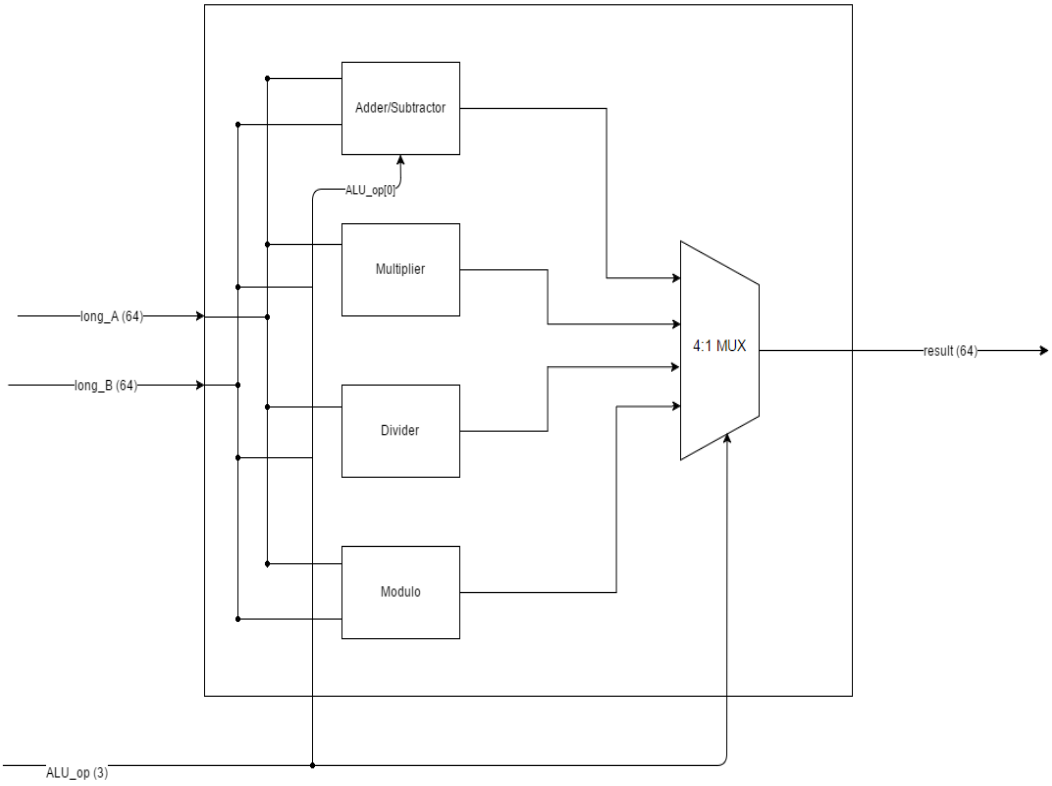


Figure 7-12 Physical implementation of arithmetic unit

Internally, the ALU generates output to all the operations. However, only one of those outputs is connected to the result bus. The arithmetic unit is programmed in such a way that a mux is used internally and one of these outputs are routed to the result bus. The ALU_op which is a 3-bit bus acts as the select lines for the mux.

Value of ALU_op	Operation
3'd00	Addition
3'd01	Multiplication
3'd02	Divide
3'd03	Modulo

3'd04	Subtraction
3'd05	XX
3'd06	XX
3'd07	XX

Table 7-2 Value on operation bus and corresponding operation

To generate all the inputs simultaneously, the arithmetic unit is internally divided into (a) an adder/subtractor, (b) a multiplier, (c) a divide and (d) a remainder generator. The adder/subtractor unit is controlled by the 3rd bit of the ALU_op bus. If the signal is zero, the unit will output the sum of the two inputs, else if the bit is set it will produce the difference.

Logical Unit

The logical unit in the JBEE is labeled as the bitwise_ALU, as it preforms most bitwise operations. It performs (a) logical AND, (b) logical OR, (c) logical XOR, (d) logical shift right, (e) arithmetic shift right and (f) logical shift left. Just as the arithmetic unit, the input to the logical unit are the two 64-bit register, long_A and long_B. As the registers are 64-bit wide all operations long, int, short or byte are performed in a single cycle. The output of the logical unit is fed to the bitwise_result bus.

Unlike the arithmetic unit, the logical unit depending on the bitwise_ALU_op input generates only a single output. This generated result is directly connected to the output bus. The bitwise_ALU_op indicates to the logical unit which output is to be generated.

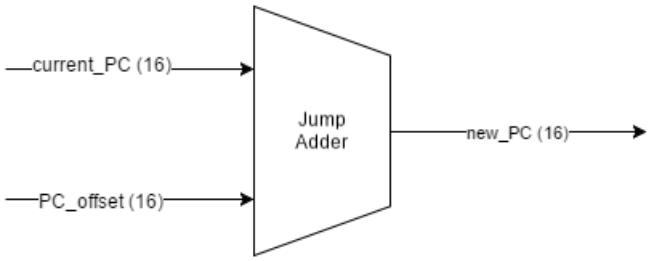


Figure 7-13 Physical implementation of jump adder

Value of bitwise_ALU_op	Operation
3'd00	logical left shift
3'd01	logical right shift
3'd02	arithmetic right shift
3'd03	logical AND
3'd04	logical OR
3'd05	logical XOR
3'd06	XX
3'd07	XX

Table 7-3 Value on operation bus and corresponding operation

Jump Adder

As the name indicates, the jump adder is used to calculate the address to which the PC should be updated to in the case of a branch. When a goto or a conditional jump instruction is encountered, if the branch is to be taken, the output of the jump adder is enabled. The update_PC signal indicates if the PC should be updated.

The two inputs to the jump adder are the current_PC register and the jump_offset register. The current_PC register is updated at the end of the instruction execution cycle i.e. in the done state. Depending on the size of the instruction, the current_PC is incremented by one, two, three or four bytes. The current_PC lags the PC register, which is part of the PC module, by 4. This is to account for the first two bytes which indicate the stack frame size and the next two specifying the size of the local variable array.

The PC_offset value is generated during either update_PC_state or the compare state. The offset is calculated using the second and the third byte of the instruction. To calculate the offset the Java standard specifies, left shifting branch byte one by 8, and adding branch byte two to the resulting value. The output of the jump adder, new_PC, is updated after every instruction but it is not valid until the update_PC signal is set.

Chapter 8

Instruction Execution

To better understand the working of the JBEE, it is best to look into how the instructions traverse through different states.

Example 1

Instruction: iinc [16]

Operation: Increment local variable by constant

Format: iinc <index> <constant>

Opcode: 132 (0x84)

Stack: No change

Description: The instruction increments the local variable at the index, which must be an integer. The index in the instruction is an unsigned byte which must be a valid offset in the local variable array. The constant, on the other hand, is a signed value.

States

- fetch

  The fetch stage being the first stage of the instruction execution cycle, the contents of the method ROM, pointed to by the PC register are read and updated into the instruction register.

- nop

  The first stage of the iinc instruction is the nop. This state does not perform any operation but allows the method ROM address to be updated and the new data i.e. the index to be available on the output data bus.

- byte_two

  During the byte_two state, the execute module reads the index from the method ROMs output data bus and saves it into the byte_two register. The value read is the index of the integer in the local variable array.

- read_local

  As the integer to be incremented is present in the local variable array, during the read_local state this value is copied into the long_A register.

  To read the value from the local variable array, the address corresponding to the index is calculated using the local adder. The index (read in earlier stage) acts as the local_offset, while the stack_size acts as the reference to point to the start of the local variable array. This address generated is available on the local_adder_out bus, and is fed to the second input of the stack arbitrator mux. To feed this address to the stack RAM, the stack_bus_select bus is set to 2'b01.

- byte_three

  During this stage, the constant is read from the method ROM and updated into the byte_three register. To speed up the instruction, the value read is also loaded into the long_B register, so the arithmetic operation can be performed directly.

  In this case, unlike the byte_two state, a clock cycle does not need to be wasted, as the method ROM output data bus has already been updated, during the intermediate read_local state.

- ALU_op

  During the ALU_op state, the execute module sets the ALU_op bus to 2'b00, generating the sum of long_A and long_B registers on the result bus.

- write_local

  The address to the stack RAM, was set as the address of the local variable pointed to by the index variable during the read_local state. The result generated during the ALU_op needs to be updated into the same location. Thus, the stack access submodule does not need to be updated. Instead, to be able to write to the stack RAM, the stack_write signal is set. And the result is updated into the stack_data_in register.

- done

  The done stage, being the last stage of the instruction cycle, the current_PC register is updated by 2'b10. Also, the stack access submodule is reset to the default, i.e. pointing to the stack top.

Example 2

Instruction: lastore [16]

Operation: store value into long array

Format: lastore

Opcode: 80 (0x50)

Stack: Before: ……, arrayref, index, value

After: ……

Description: The lastore instruction is used to save the value from the top of stack into an element of the local array pointed by arrayref, at the index location. The arrayref is of type reference and the index is an usigned integer. Since, the data is being written to an long array, the value is 64-bits and occupies two locations on the stack.

States

- fetch

  The fetch stage being the first stage of the instruction execution cycle, the contents of the method ROM, pointed to by the PC register are read and updated into the instruction register.

- dec_SP

  As describeds earlier, the stack in the JBEE is designed as a Pre-Decrement/Post-Increment Stack architecture. Thus, to make the stack pointer point to the value_MSB (higher 32-bits of the long data), the stack pointer is decremented. Since, the default state of the stack access subsystem is read stack section, no other modifications are made during this state.

- read_stack

  The destination of the data read during the read_stack state is determined by the iteration variable, as this state is invoked multiple times during this instruction execution cycle.

  During the first invocation of the read_stack state, the value of iteration is 2'b00 and thus, the data is read from the stack top and stored into the higher bits [32:63] of the long_A register. After the data has been read out the value of the iteration variable is incremented.

- dec_SP

  As seen in the first invocation of the dec_SP state, the SP is now decremented to point to the value_LSB as the higher bits [32:63] was popped of the stack in the last state, thus, pointing to the valid top of stack.

- read_stack

  As the value of the iteration variable during this state is 2’b01, the value read from the stack RAM is stored into the LSB [0:31] of the long_A register. The long_A now contains the complete value to be written into the array.

  Also, the iteration register is incremented, and now contains the value 2’b10.

- dec_SP

  The stack pointer is decremented again to point to index value, is now the valid top of stack.

- read_stack

  During this stage, the iteration value is 2’b10. The value, thus, read from this stack RAM is stored into the array_index register. The index as it points to an element in the array on the heap is used as an operand to the heap adder.

  The iteration variable is incremented to 2’b11.

- dec_SP

  During the final iteration of the dec_SP of this instruction execution cycle, the stack pointer is decremented to point further, to the valid top of stack. The stack pointer now points to the arrayref.

  Note: As the stack is the top of the memory, and in certain cases there may not be any element on the stack left, the stack pointer may roll over to the highest address (0x3f).

- read_stack

  During the final iteration of the read_stack, the register variable is at 2’b11. The arrayref is read from the stack and then stored into the array_ptr register.

As the array_ptr points to the start of the register and the array_index points to the precise element in the array, the exact memory location to be written to is obtained. The iteration value is also incremented and rolls around to 2'b00.

- write_heap

  During the write_heap state, the stack access subsystem is set up to write to the heap section of the stack RAM. The stack_select_bus is set to 2'b10, so as to enable the heap adder output to connect to the stack_bus. Since, the write_heap state is also invoked more than once, the iteration register value is checked before writing to the array.

  The heap is written the LSB value of long_A register when the iteration value is 2'b00.

- handle_long

  The handle_long is a special state, invoked while writing a long integer. During this state, the heap pointer is incremented, to point to the next memory location, to store the MSB of the long value. This is done by incrementing the array_index register by one.

- write_heap

  During this stage the long_A MSB to written to the array element. Thus, the long integer is completely transferred from the stack to the heap at the required memory locations.

- done

The done stage, being the last stage of the instruction cycle, the current_PC register is updated by 2'b10. Also, the stack access submodule is reset to the default, i.e. pointing to the stack top.

Chapter 9

Results

Implementation

The hardware used for the project is the Altera's DE1 development board. Since, in the project a processor has been developed only a limited hardware elements provided by the board are used. The JBEE itself is programmed into the Cyclone II FPGA chip. The seven-segment display is used to show the value of the program counter register. The red LEDs on the board are connected to the state register and thus indicates the current stages of the processor at a certain point in time.

The green LEDs are linked to the return_inst state. When the state is invoked, the LEDs are set indicating the program execution has finished.

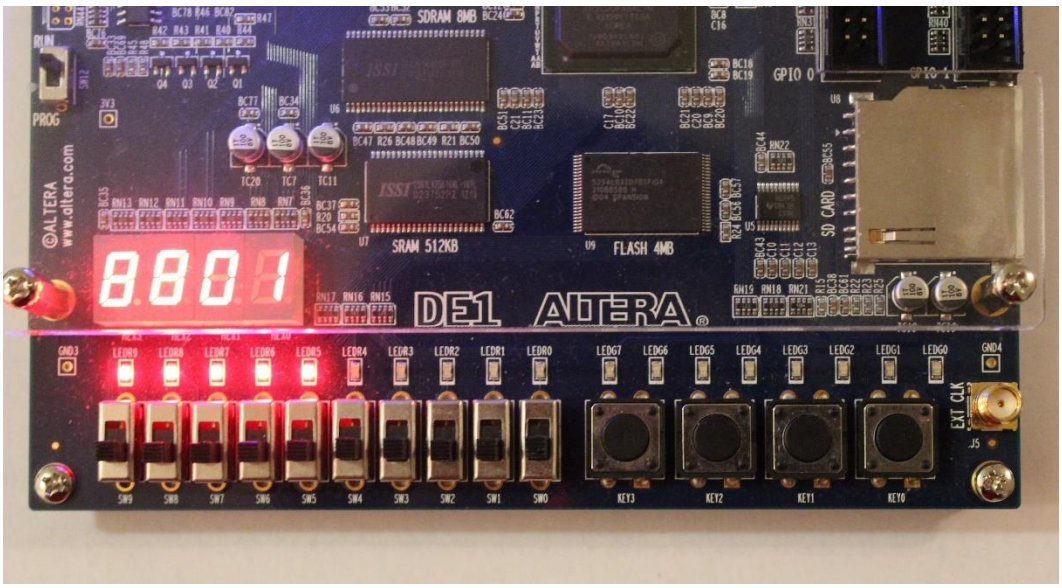


Figure 9-1 Image showing the PC (seven-segment LEDs) and state (red LEDs) on the DE1 board.

Simulation

To get a better understanding of how things work inside the JBEE, its best to observe the simulation results. The simulation results are obtained via. ModelSim, a multi-language HDL simulation environment by MentorGraphics [35].

Since it is overwhelming to go over the entire execution of the program, the waveforms have been broken down into sections.

Note 1:

The M4F memory block in the Altera Cyclone II, update the value on the output bus at the rising edge of the clock, while values are written into the memory on the falling edge of the clock.

Note 2:

The Java method stored in the code attribute is presented in such a way that the first two bytes contain the maximum size of the stack frame, while the next two bytes contain the size of the local variable array for the corresponding method.

Waveform 1

In the waveforms fig. 9-1, the first five clock cycles the program goes through while execution. These waveforms are similar for any program being executed, the only difference being the values updated into the stack_size register and the Local Pointer (LP) register.

The initial values are assigned to the registers and buses, at the start of the execution. These initial values include:

REGISTER/SIGNAL	**INITIAL VALUES**
PC	8’b00000000
stack_size	16'b0
LP	16'b0
HP	16'b0
stack_ptr	6'h3f

stack_add_sub	1'b1
stack_add_enable	1'b1
iteration	2'b00
current_PC	16'b0
update_PC	1'b0
return_inst	1'b0
pool_pointer	6'b0
LED	10'b1111111111

Table 9-1 Signals/Registers and corresponding initial values

On the rising edge of the first clock cycle, the PC register is incremented from (0x0), to (0x1). The value corresponding to the PC memory location is obtained during the next rising edge of the clock. Hence, on the second rising edge of the clock, the value (0x00) is obtained from the PC register which corresponds to the memory location (0x0). This value is then written into the lower eight bits of the stack_size register.

On the next clock cycle, when the method ROM outputs the value corresponding to memory location (0x1), which in this case is (0x02). This value is then updated into the higher eight bits of the stack_size register. The update value can be seen in the stack_size register, as its contents change from (0x00) to (0x02).

Similarly, on the following clock cycle, the value obtained from the method ROM corresponds to the PC (0x2) and is (0x00). This value is written into the lower eight bits of the LP register. The next value read from the PC register, corresponding to memory location (0x3) is (0x06). This value is then updated into the higher byte of the LP register. This is evident, as the value in the register changes from (0x00) to (0x06).

Also, as the processor is initializing the value of the memory register, the state register contains the value (0x39). This indicates the processor is in exception state, no data is loaded into the IR register.

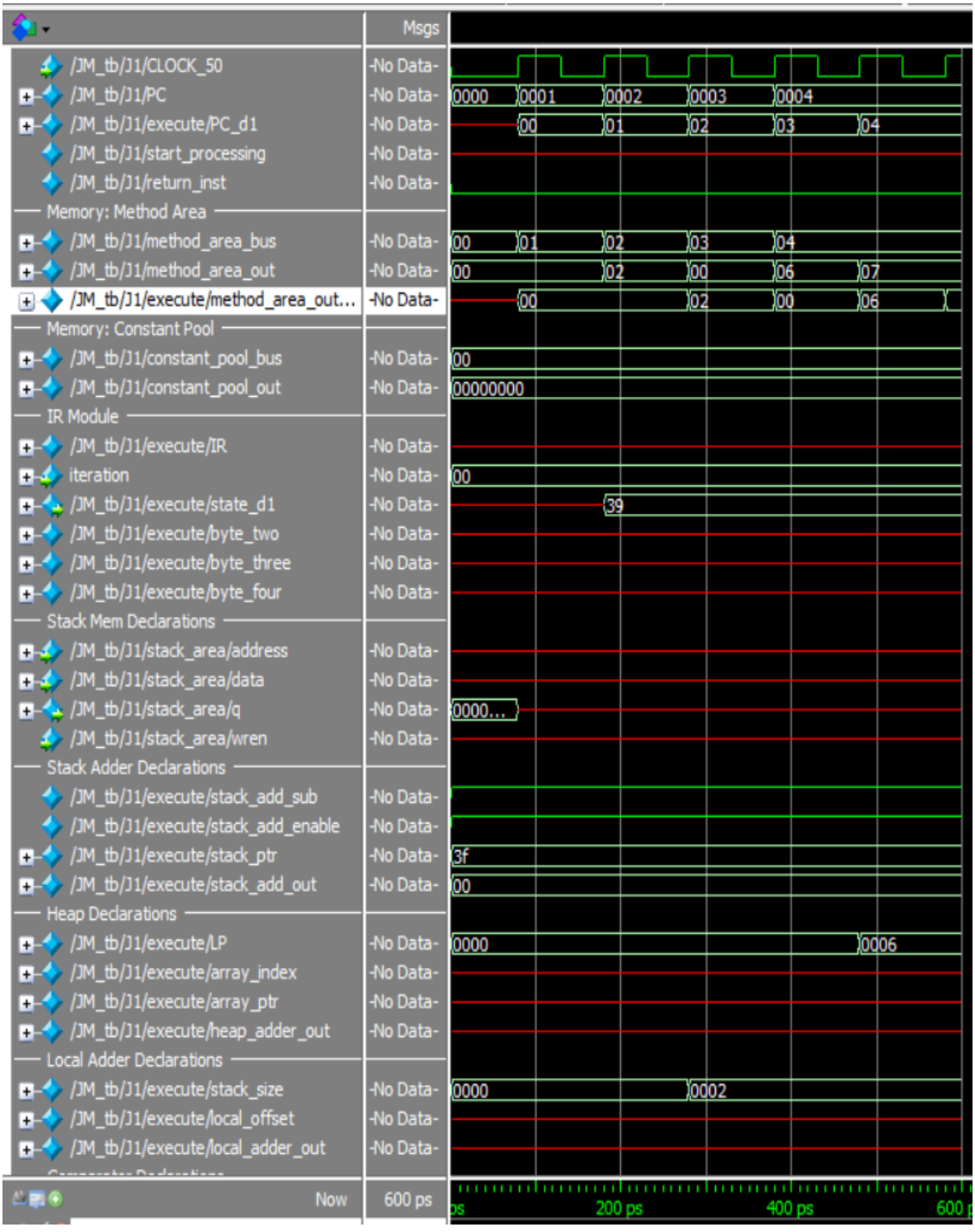


Figure 9-2 Waveform depicting initialization of memory registers

Waveform 2

The first four bytes of the method ROM has the contains values for the memory registers. Following these four bytes, the fifth byte contains the opcode of the first instruction. When the address of the PC is incremented to (0x4), the data on the output bus of the method ROM is (0x07). The succeeding clock cycle, the start_processing signal is set and the opcode is loaded into the IR register. This changes the state of the machine to (0x00) i.e. fetch, and the IR checks if the opcode is a valid.

The state register, as can be seen from the waveform below, transitions at the falling edge of the clock. Each

The opcode (0x07), loaded in the IR indicates the instruction is an iconst_4. The state through which the iconst_4 instruction transitions is:

- fetch

  As mentioned above, when the start_processing signal is set, the processor changes state from exception to fetch. During the fetch state, the method ROM output data bus is read and the value is loaded into the IR register. Since, no state in the instruction is repeated, the iteration variable remains unused.

- write_stack

  The iconst_4 instructions loads an integer four onto the stack. Since, this value is not being read from memory or loaded from a register, it is generated internally, and loaded in the stack_data_in_register. The stack_write_en signal is set, so as to indicate that the stack RAM is being written to.

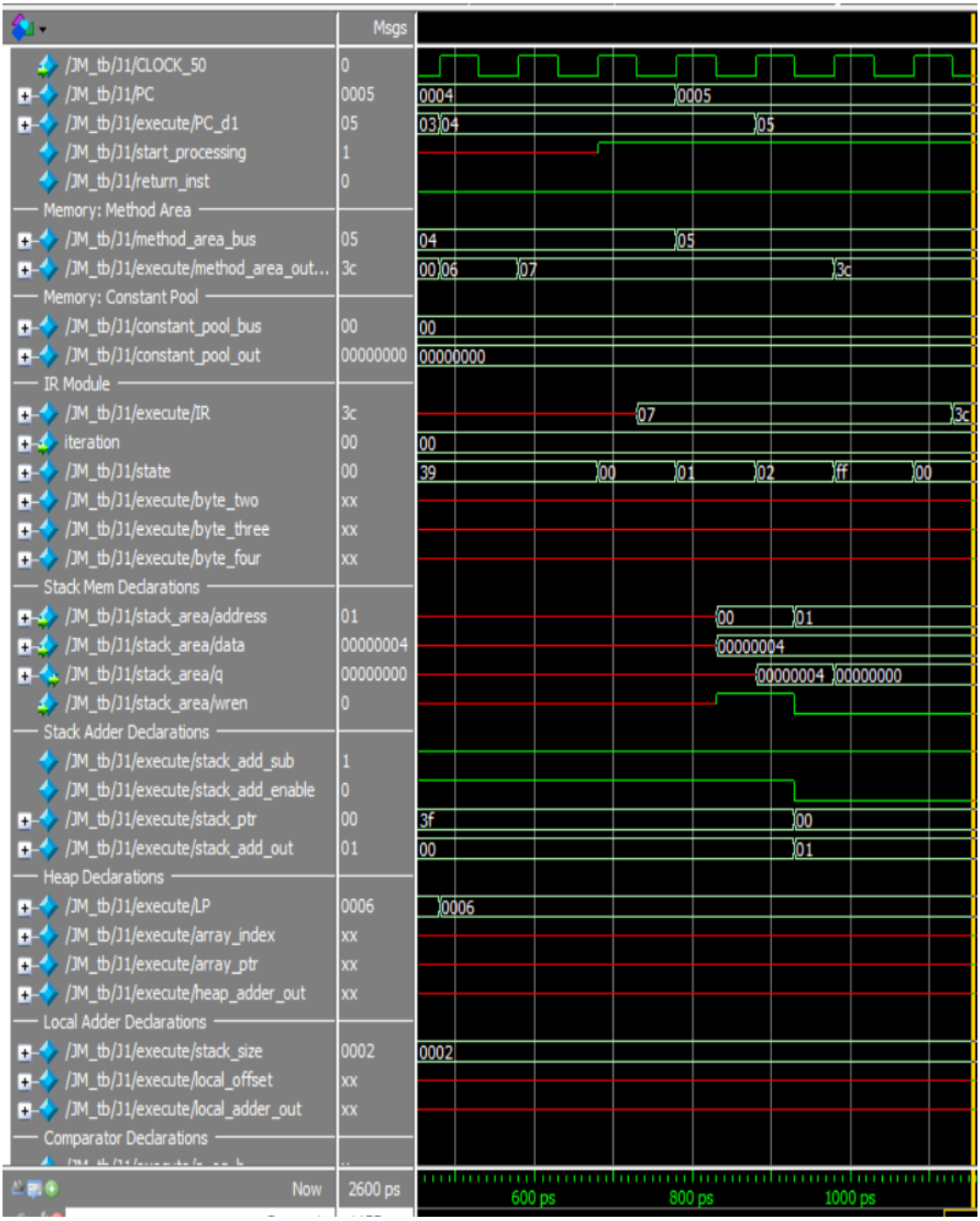


Figure 9-3 Waveforms indicating the start of processing

Also, evident from the waveforms, the stack_ptr register points to the highest location I the memory i.e. 8'h3f. Though it may seem, the stack_ptr points to an invalid address, the memory location written to is one added to this address, as the output of the stack_access_adder, is the actual address that is being written to, while the stack_ptr register points to the top of stack.

- inc_SP

  The inc_SP state, increments to the value of the stack_ptr variable. Since, the stack RAM was written to during the preceding stage, the top of stack incremented, causing the stack_ptr to hold an invalid address. The stack_add_sub signal is enabled, indicating the stack_ptr needs to be incremented. And to enable the adder to do so, the stack_add_enable signal is complemented.

- done

  As this is not a branch or a jump instruction, the only action performed during this stage is increment of the current_PC register, and the iteration register is reset to its default value, 2'b00.

Waveform 3

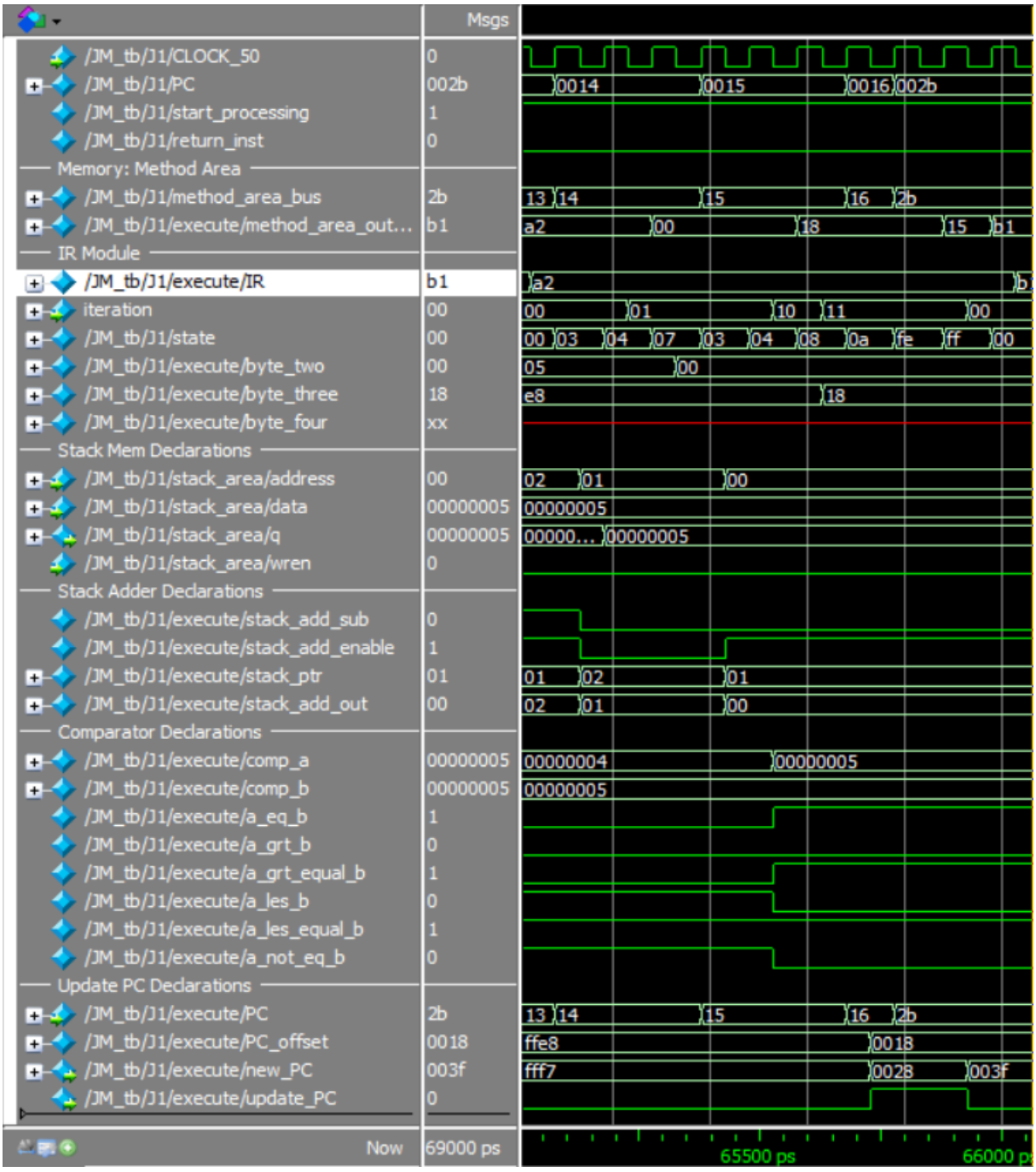


Figure 9-4 Waveforms showing PC update

The waveform above, show the execution of a conditional branch instruction. The instruction loaded into the IR is if_icmpge, and has the opcode (0xa2). The instruction indicates, if value1 is greater than or equal to value2, the branch must be taken.

The instruction contains two additional bytes, branchoffset1 and branchoffset2, that are used to generate the offset address. These are unsigned both unsigned byte numbers. This offset address is then added to the current_PC register, to obtain the jump address. Depending on whether the jump is to be taken or not, the address is loaded into the PC modules register.

The two values to be compared, value1 and value2, are present on the stack register and must be transferred to the comp_A and comp_B registers. The first two states, i.e. dec_SP and read_stack, pop the value2 off the stack top and stores it in register comp_B. The iteration register value is also incremented during the read_stack state, as two values need to be popped of the stack, and multiple invocation of the dec_SP and read_stack states is necessary.

The following state, which is the byte_two state, the method_ROM output data bus is read and the branchoffset1 value is loaded into the byte_two register. As soon as the byte_two state is reached, the PC module increments the PC register. The PC now points to the location of byteoffset2.

For the next two stages, dec_SP and read_stack, the value of the iteration variable is 2'b01. When value2 is read from the top of stack, the value is written into register comp_A. The iteration variable is incremented again to 2'b10.

During the next stage, byte_three, the byteoffset2 value is read and loaded into the byte_three register. The complete offset value is then calculated, using the formula branchbyte1 << 8 + branchbyte2. The value generated is of two complement notation. If

the value is negative then the jump is backwards, compared to the current PC value, else it's a forward jump.

As the values were transferred to the comparator registers as they were popped from the stack, during the compare stage, the required comparator output is checked. If the output is set or high, the branch is to be taken. In this case, as the a_grt_equal_b signal is set, the branch is to be taken. To indicate to the PC_module to update the address of the PC with the new_PC data, the update_PC signal is set. The current_PC is update by one byte as well, to account for the if_icmpge instruction itself being read.

The compare state is followed by an NOP stage, to allow the PC to be loaded and the address to be fed to the method ROM.

During the final state, i.e. done, the iteration register is set back to 2'b00. No action is taken with the current_PC register as the value was updated in the compare state.

Waveform 4

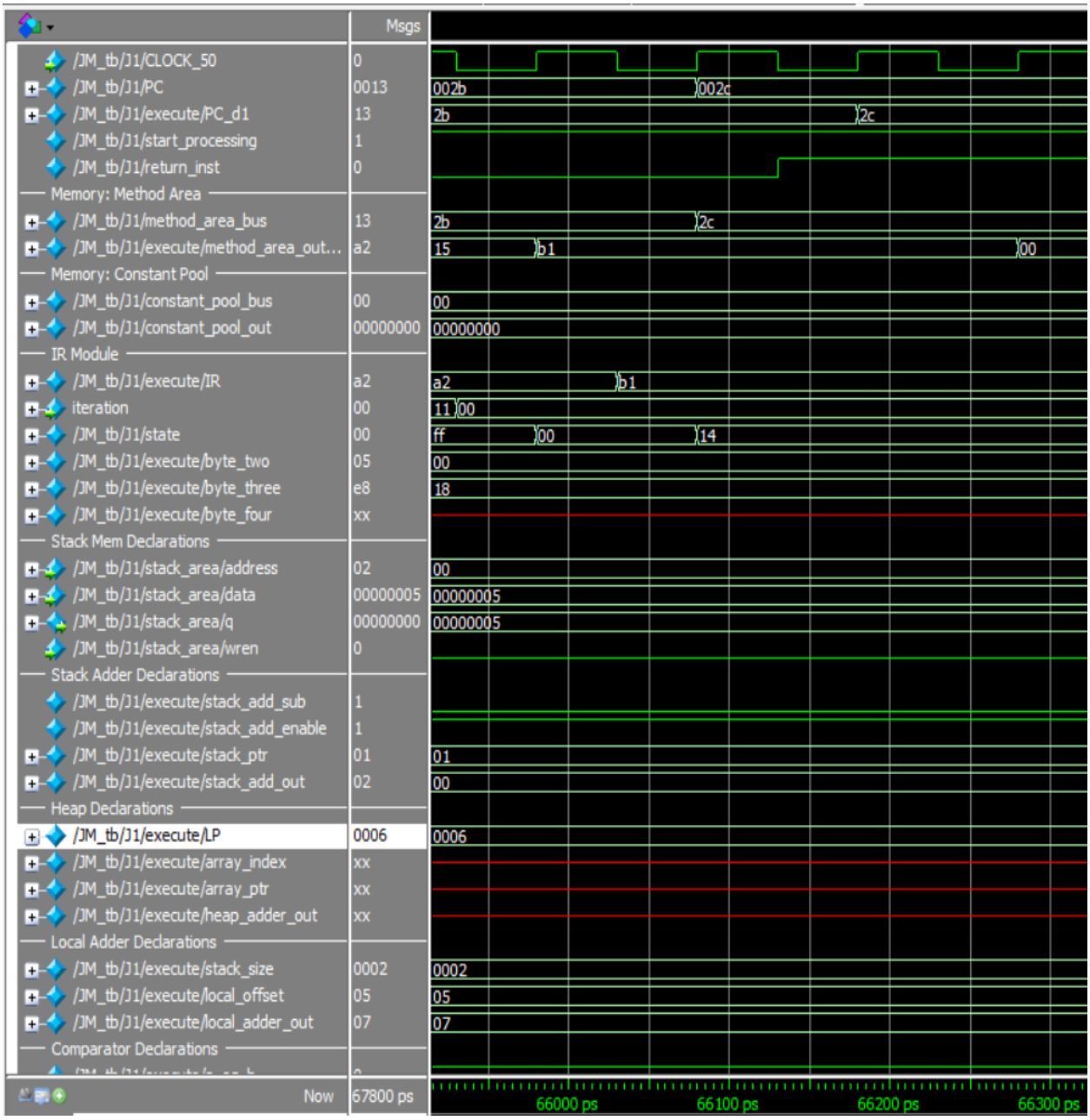


Figure 9-5 Waveform showing end of execution

The last instruction executed in a program, is always the return (0xb1) instruction. As seen in the waveform, during the fetch stage the method ROM output data bus is read and loaded into the IR register. The execution cycle of the IR instruction only contains the special state, return_inst. Once the processor enters the return_state, the state does not change as the program execution stops.

During the return_inst state, the return_inst signal is set. Due to a positive transition on the signal, triggers the LEDs to be set.

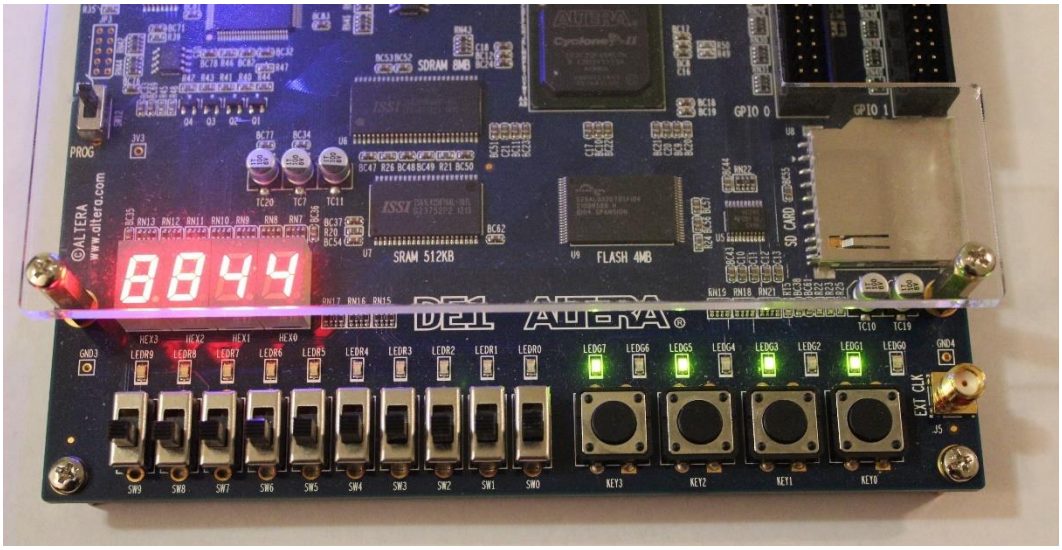


Figure 9-6 Image showing end of execution and final PC.

Also, since the processor enters the fetch state, the PC module increments the PC register. This leads to the PC pointing to an address with an invalid instruction This will present an invalid address on the seven-segment display, and will be offset by 1’b1, as it is linked to the PC address from PC module. But as the data is not loaded into the IR, this does not interfere with the execution of the program.

Appendix A

List of Instructions Implemented And State Transition

The following list gives all the instructions implemented and shows the states through which they transition:

Note: Since, all instructions begin with the Fetch stage it is excluded from the state transition list.

iconst_value:

- write_stack
- inc_SP
- done

istore index:

- dec_SP
- read_stack
- byte_two
- write_local
- done

istore_value:

- dec_SP
- read_stack
- write_local
- done

iload index:

- nop
- byte_two
- read_local
- write_stack

- inc_SP
- done

iload_value:

- read_local
- write_stack
- inc_SP
- done

dup:

- read_stack
- write_stack
- inc_SP
- done

if_compare:

- dec_SP
- read_stack
- byte_two
- dec_SP
- read_stack
- byte_three
- compare
- nop
- done

goto:

- nop

- byte_two
- nop
- byte_three
- update_PC
- done

iadd: (all integer arithmetic instructions)

- dec_SP
- read_stack
- dec_SP
- read_stack
- ALU_op
- write_stack
- inc_SP
- done

iinc:

- nop
- byte_two
- read_local
- byte_three
- ALU_op
- write_local
- done

newarray:

- dec_SP

- read_stack
- byte_two
- write_stack
- inc_SP
- inc_HP
- done

iastore:

- dec_SP
- read_stack
- dec_SP
- read_stack
- dec_SP
- read_stack
- write_heap
- done

bipush:

- nop
- byte_two
- write_stack
- inc_SP
- done

sipush:

- nop
- byte_two

- nop
- byte_three
- write_stack
- inc_SP
- done

ldc:

- nop
- byte_two
- read_pool
- write_stack
- inc_SP
- done

ldc_w:

- nop
- byte_two
- nop
- byte_three
- read_pool
- write_stack
- inc_SP
- done

ldc2_w:

- nop
- byte_two

- nop
- byte_three
- read_pool
- write_stack
- inc_SP
- read_pool
- write_stack
- inc_SP
- done

12l:

- dec_SP
- read_stack
- write_stack
- inc_SP
- write_stack
- inc_SP
- done

lastore:

- dec_SP
- read_stack
- dec_SP
- read_stack
- dec_SP
- read_stack

- dec_SP
- read_stack
- write_heap
- handle_long
- write_heap
- done

lstore_value:

- dec_SP
- read_stack
- write_local
- dec_SP
- read_stack
- write_local
- done

lstore index:

- dec_SP
- read_stack
- byte_two
- write_local
- dec_SP
- read_stack
- write_local
- done

astore:

- dec_SP
- read_stack
- byte_two
- write_local
- done

aload:

- nop
- byte_two
- read_local
- write_stack
- inc_SP
- done

lload_value:

- read_local
- write_stack
- inc_SP
- read_local
- write_stack
- inc_SP
- done

lload index:

- nop
- byte_two

- read_local
- write_stack
- inc_SP
- read_local
- write_stack
- inc_SP
- done

ladd: (all long integer arithmetic instructions)

- dec_SP
- read_stack
- dec_SP
- read_stack
- dec_SP
- read_stack
- dec_SP
- read_stack
- ALU_op
- write_stack
- inc_SP
- write_stack
- inc_SP
- done

lneg:

- dec_SP

- read_stack
- dec_SP
- read_stack
- ALU_op
- write_stack
- inc_SP
- write_stack
- inc_SP
- done

ishl:

- dec_SP
- read_stack
- dec_SP
- read_stack
- ALU2_op
- write_stack
- inc_SP
- done

lshl:

- dec_SP
- read_stack
- dec_SP
- read_stack
- dec_SP

- read_stack
- ALU2_op
- write_stack
- inc_SP
- write_stack
- inc_SP
- done

land:

- dec_SP
- read_stack
- dec_SP
- read_stack
- dec_SP
- read_stack
- dec_SP
- read_stack
- ALU2_op
- write_stack
- inc_SP
- write_stack
- inc_SP
- done

References

[1] A. Aaby, "Introduction," 1998. [Online]. Available: http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm.

[2] V. Beal, "What is programming language? Webopedia definition,". [Online]. Available: http://www.webopedia.com/TERM/P/programming_language.html.

[3] Divestopedia and S. Institute, "What is a programming language? - definition from Techopedia," Techopedia.com, 2016. [Online]. Available: https://www.techopedia.com/definition/24815/programming-language.

[4] Sammet, J. E. (1972). Programming languages. Communications of the ACM, 15(7), 601–610. doi:10.1145/361454.361485

[5] Rosenblum, M. (2004). The Reincarnation of virtual machines. Queue, 2(5), 34. doi:10.1145/1016998.1017000

[6] Toal, R. Pltypes. Retrieved December 6, 2016, from http://cs.lmu.edu/~ray/notes/pltypes/

[7] Novoselsky, A., and Karun, K. (2011, April 15). XSLTVM — an XSLT virtual machine [Anguel Novoselsky, K. Karun]. Retrieved December 6, 2016, from https://dret.net/biblio/reference/nov00

[8] Comparison of application virtualization software (2016). . In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Comparison_of_application_virtualization_software

[9] "Chapter 1 introduction to the java programming environment (JDK 1.1 for Solaris developer’s guide)," in Oracle, 2010. [Online]. Available: https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqb/index.html.

[10] "Apache commons BCEL™ – the java virtual machine," in Apache Commons, 2004. [Online]. Available: https://commons.apache.org/proper/commons-bcel/manual/jvm.html.

[11] "Oracle technology network," in Oracle. [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index.html.

[12] "Section 1.4. Java's magic: Bytecode, java virtual machine, JIT, JRE and JDK," in Java School, Java School, 2013. [Online]. Available: https://j4school.wordpress.com/java-tutorials/core-java/introduction-to-java/java-magic-bytecode-java-virtual-machine-jit-jre-jdk/.

[13] B. Venners, "Java virtual machine's internal architecture," in artima developer, 1996. [Online]. Available: http://www.artima.com/insidejvm/ed2/jvm2.html.

[14] R. Ramachandran, "JVM (java virtual machine) architecture - tutorial," in YouTube, YouTube, 2015. [Online]. Available: https://www.youtube.com/watch?v=ZBJ0u9MaKtM.

[15] "Chapter 4. The class File Format," in Oracle. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html.

[16] "Chapter 6. The Java Virtual Machine Instruction Set," in Oracle. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.2.

[17] M. Dahm, "Byte Code Engineering with the BCEL API," Freie Universitat Berlin, Berlin, Germany, Apr. 03, 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.1710andrep=rep1andtype=pdf.

[18] R. ROOSTA, "Programmable Logic Devices," in ECE595. Northridge, CA.

[19] A. Kent, "A Texas Instruments Application Report: MOS programmable logic arrays," Texas Instruments, California, Oct. 1970.

[20] M. Ferdjallah, Introduction to digital systems: Modeling, synthesis, and simulation using VHDL [Book]. Suffolk, Virginia: John Wiley and Sons, 2011. [Online]. Available: https://www.safaribooksonline.com/library/view/introduction-to-digital/9780470900550/.

[21] K. Kant, "GAL - generic array logic," in *FPGA Central*, 2011. [Online]. Available: http://www.fpgacentral.com/pld-types/gal-generic-array-logic.

[22] W. N. Wan Ibrahim, "Multiplexers, Decoders and Programmable Logic Devices," in Slideshare, 2013. [Online]. Available: http://www.slideshare.net/WanNurdiana/mux-decod-pld2vs2.

[23] "CPLD," in Xilinx. [Online]. Available: http://www.xilinx.com/cpld/.

[24] "FPGA fundamentals," in National Instruments, 2008. [Online]. Available: http://www.ni.com/white-paper/6983/en/.

[25] "DE1 Board Introduction," 2006. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=EnglishandCategoryNo=165andNo=83andPartNo=4.

[26] "DE main boards - Cyclone - Altera DE1 board," in Terasic Technologies, 2006. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=83.

[27] W. Puffitsch and M. Schoeberl, "PicoJava-iI in an FPGA," ACM, 2007, pp. 213–221. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1288940.1288972.

[28] Cision, "IMSYS launches Cjip the ‘java system on a chip’ at JavaOne," News Powered by Cision, 2000. [Online]. Available: http://news.cision.com/se/imsys/r/imsys-launches-cjip--the--java-system-on-a-chip--at-javaone,e24234.

[29] aj. Systems, "AJile systems introduces the aJ-100, the world’s First single chip java Microcontroller with an embedded real-time kernel," PR Newswire, 2000. [Online]. Available: http://www.prnewswire.com/news-releases/ajile-systems-introduces-the-aj-100-the-worlds-first-single-chip-java-microcontroller-with-an-embedded-real-time-kernel-73579227.html.

[30] "aJ-100TM Real-time Low Power JavaTM Processor," 2000. [Online]. Available: https://www.digchip.com/datasheets/download_datasheet.php?id=126782andpart-number=aJ-100.

[31] R. Zulauf, "Komodo-Mikrocontroller," in zulauf-online, 2000. [Online]. Available: http://zulauf-online.name/da/node56.htm.

[32] S. Uhrig, J. Mische, and T. Ungerer, "An IP Core for Embedded Java Systems," in *University of Augsburg*, Augsburg, Germany, 2007. [Online]. Available: http://samos-conference.com/Resources_Samos_Websites/Proceedings_Repository_SAMOS/2007/Files/2007-WS-28.pdf.

[33] "Memory Initialization file (.mif) definition," in Altera, 2005. [Online]. Available: http://quartushelp.altera.com/15.0/mergedProjects/reference/glossary/def_mif.htm.

[34] *Quartus II*. Altera, 2016. Print.

[35] T. Granberg, Handbook of digital techniques for high-speed design. India: Pearson Education, 2007. [Online]. Available: https://books.google.com/books?id=Eqgrqktkk1YCandpg=PA606anddq=ModelSimandhl=enandsa=X#v=onepageandq=ModelSimandf=false.