INVESTIGATING THE IMPLEMENTATION OF MAXINE VM ON THE ARMV8-A 64-BIT ARCHITECTURE

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2015

By
Jiaqi Liu
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

Java Virtual Machines (JVMs) enable programming languages, not only Java, to be executed on various platforms without modification. Among many prominent JVMs, The Maxine VM provides an extensible research platform for high level managed languages and their compilation, due to the proactive use of software abstraction and a modular structure.

This project is to investigate implementing Maxine on the state-of-the-art ARMv8 64-bit architecture, which is helpful for high level language research on this new RISC architecture. This thesis gives a progressive description of the project, involving the implementation of two fundamental components, an architecture representation and an assembler system, of the Maxine VM on ARMv8 along with its research context, design and evaluation.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I am deeply indebted to my supervisor, research fellows in the University of Manchester and many other people for their clear guidance, thoughtful opinion and optimistic encouragement. This project would not have been done without their help throughout the semester.

Dr. Mikel Luján, my supervisor, gave me great freedom to explore many fields of Java virtual machines and helped me a lot when I was planning the project and writing the thesis. Outside this project, Mikel also gave us inspiring and helpful MSc courses on parallel computing and multi-core processor. Dr. Christos Kotselidis offered me great help of both introducing me the structure of the large Maxine VM code base and providing me the unit test environment. I am very thankful for his responsive feedback and real-time help. Dr. Andy Nisbet gave me thoughtful advice on learning Maxine VM and how to organise the thesis. Andy also provided insightful opinions on presenting the result of the project.

I would also like to thank Guillermo Callaghan and Wei Huang for their giving me useful information when I was lost in the vast 'abyss' of materials and documents about ARMv8. They also helped me improve my English language skills and gave me useful feedback on my thesis draft.

Chapter 1

Introduction

1.1 Project Context

Java, referred to as both a programming language and a software platform, is one of the most popular and successful software systems [TIO15]. While big data processing is gaining its importance nowadays, Java also plays an important role in this trend. Distributed data processing software frameworks written in Java, such as Apache Hadoop, have proven their power in high-performance computing over a large scale of data [Had15].

The popularity of Java largely owes to its feature of being platform independent and robust. The runtime environment of Java has been implemented on various CPU

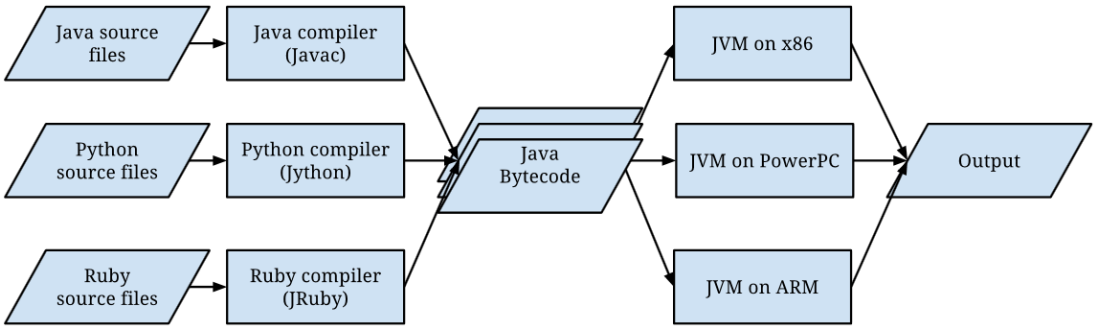

Figure 1.1: JVMs on different platforms.

architectures and operating systems, which gives Java programs great portability. As the core part of the Java runtime, Java virtual machines (JVMs) execute Java bytecode, usually compiled from Java source files, on different platforms and guarantee the correctness of the execution. Figure 1.1 shows that different languages can be compiled to Java bytecode and then be executed by JVMs on different platforms.

In the last two decades, many JVM implementations have been implemented [SSB12]. Among them, the Maxine virtual machine is mainly targeted at the research for Java language and compilers. Maxine is entirely written in Java (a metacircular JVM) with high level of abstraction [WHVDV+13]. Maxine VM can use whatever optimising compilers as long as they have implemented the compiler-runtime-interface (CRI, a compiler interface that defines how a compiler should behave in order to be used in different JVMs). Thus, Maxine provides an ideal environment for high level language and compilers research.

The ARM architecture is a reduced-instruction-set-computing (RISC) architecture. In terms of the number of annually shipped processors, it accounts for the largest microprocessor market share in recent years [Shi14]. Most handheld devices, including smartphones and tablet PCs, are equipped with ARM processors. Meanwhile, ARM processors provide relatively better energy efficiency compared with complex-instruction-set-computing (CISC) ones. Recently, 64-bit ARM architecture (ARMv8 or AArch64) has been introduced [ARM11]. Together with the ARM big.LITTLE heterogeneous architecture [ZR13], which was announced in 2011, ARMv8 provides a more efficient way of energy saving computing.

However, a problem appears if researchers are looking for a JVM for Java language research on the ARMv8 architecture. The Maxine VM has no official support for ARMv8 (only the support for the x86_64 architecture is officially provided).

To gain knowledge of the performance of just-in-time (JIT) compilers and managed runtime environments, it is important to investigate them on state-of-the-art processor platforms. Therefore, the necessity for porting Maxine to ARMv8 is clear. Once succeeding in the port, developers will be offered an excellent opportunity for the research of compilers and managed runtime environment on ARMs 64-bit architecture.

1.2 Aim and Objectives

Simply stated, the aim of this project is to investigate the implementation of Maxine VM on the ARMv8 architecture. As will be discussed in detail in Section 2.1, Maxine VM consists of well-defined modules that make it possible to be ported to other architectures other than x86_64.

To be more detailed, the objectives of this project are listed as below:

- Setting up an environment for the Maxine VM and the Graal compiler that is needed as the optimising JIT compiler in Maxine. (The combination of Maxine and JIT compilers will be introduced in later sections.)
- Implementing ARMv8 architecture representation in Maxine and Graal source code. The representation consists of register organisation and architecture features representations, such as endianness and memory model.
- Implementing the Maxine ARMv8 assembler system. The assembler system assembles ARMv8 assembly instructions to target machine code.

1.3 Overview

Chapter 2 presents background knowledge required to give clear research context. The following topics are covered:

- Java virtual machines.
- Maxine VM and its components.
- ARMv8 architecture

Chapter 3 describes the implementation of two key modules of the Maxine VM on ARMv8:

- ARMv8 architecture representation.

- Maxine ARMv8 assembler system, including an assembler and a macro assembler.

Chapter 4 introduces how the result are evaluated using a unit test framework and the ARMv8 GNU tool chain.

Chapter 5 discusses conclusions of this project and give some suggested future work related to this project.

Chapter 2

Background

2.1 Java Virtual Machines

Ranging from hand-held devices to large-scale data processing servers, the application of the Java technology platform in different areas has significantly promoted modern software world. JVMs have contributed to this success by providing Java with the cross-platform feature. As discussed in Section 1.1, JVMs have been implemented on various architectures. They take Java bytecode as input and execute programs on different platforms without the need of modifying Java source files.

A running JVM is a process that reads the bytecode instructions and interpret or compile them into target machine code. More details about JVM interpretation and compilation will be discussed in Section 2.2.

The expected behaviour of any JVM is defined by the Java Virtual Machine Specification [LYBB14]. Not focusing on how a JVM is implemented, the specification only describes what needs to implement to be qualified as a JVM. As a result, numerous JVM implementations have taken place on various platforms: Oracle Hotspot VM [Ope15], Maxine VM [Sim11b] [Sim08] and Apache Harmony VM [Apa05] to name but a few.

The aim of most JVMs is to obtain high execution performance. Taking Hotspot as an example, to achieve this goal, the source code of Hotspot is relatively complex with

many optimisation tricks that often confuse high level language researchers [Šev08]. Adding an extensional feature or function in such complex software systems is often time-demanding and error-prone.

2.2 Maxine VM

The Maxine VM is a research virtual machine implementation mainly aimed at managed language research. Maxine is entirely written in Java, which can be called as a meta-circular JVM (Java programs executing on a JVM written in Java). It has a high level modular design with a clear logic for how different modules of the VM work together.

Maxine is a platform on which developers can carry out managed language and compiler research since it is feasible to write extensions and extra add-on modules for such a highly modular virtual machine. Further details are discussed in Section 2.2.1.

2.2.1 Maxine VM Overview

Figure 2.1 shows the overall structure of Maxine. The top part of the figure consists of the Java bytecode processing modules. Java class files (bytecode) are firstly loaded and verified. Then they are compiled by the baseline compiler (T1X) or by the optimising compiler (Graal or C1X depending on given parameters). These compilers will be further discussed in the next section.

The lower part of the figure consists of modules that manage the execution of compiled code. Memory management (garbage collection) and thread scheduling are two major modules. Written in Java, Maxine is equipped with the ability of utilising high level abstraction and decoupled design to make it an extensible virtual machine for different target architectures [WHVDV$^+$13]. Its coverage of the JVM specification is nearly complete which makes it able to execute Java bytecode compiled from standard Java Development Kit (JDK) compilers.

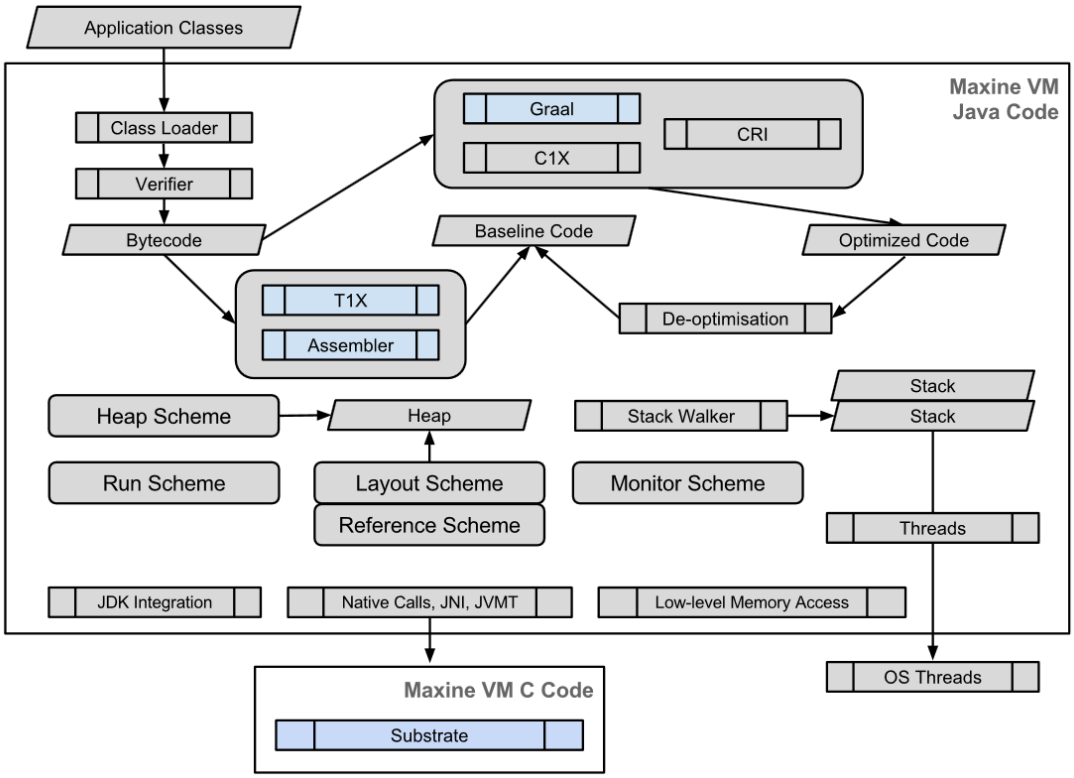

Figure 2.1: Structure of the Maxine VM.

2.2.2 Assembler, Interpreter and Compiler

When the first version of Java Development Kit (JDK 1.0) was released in 1996 [OG01], it only provided a pure interpretive execution virtual machine (Sun classic VM). In other words, at that time Java bytecode could only be interpreted, not compiled [Ven96]. As a result, Java suffered from a performance penalty, often much slower than native languages such as C or Pascal [Blu96].

To improve the performance of Java programs, in 1998 a virtual machine named Hotspot was released along with JDK 1.2 and then later became the default VM for JDK and JRE (Java Runtime Environment) [KWM+08]. The Hotspot VM comes with a just-in-time (JIT) compiler named C1 (client compiler) which can compile frequently executed code (hotspot code) into optimised machine code to avoid repeated interpretation.

However, interpreters are not abandoned. One important reason is that converting

Java bytecode to machine code via interpretation is faster than via compilation (although the interpreted code is often slower due to the lack of optimisation). Thus, to minimise the loading time of a piece of a program, an interpreter is used to convert the bytecode instructions into machine code without having to wait for the optimisation that a JIT compiler performs.

After a piece of Java bytecode is loaded by the JVM, it is firstly interpreted and then executed. If later this code is detected as a hotspot because it is repeatedly executed, the optimising compiler will compile it into optimised machine code. According to [Gou11], hotspot code accounts for 80% - 90% of a programs execution time while for only around 20% of the code size. So it is effective only to compile hotspot code because it gives a good balance of loading time and executing time.

Assembler

As discussed in Section 2.2.1, Maxine uses both T1X and Graal to compile Java bytecode. When T1X is used, the intermediate compilation result is passed to an assembler to generate machine code. This assembler is called the Maxine Assembler System (MAS) [Sim11a].

MAS has official support only for the AMD64 (x86_64) architecture. However, it contains several levels of implementation which make it easy to be adapted to new target platforms. In this project, the ARMv8 implementation for the assembler is carried out.

Listing 2.1: Templates used by T1X compiler.

```
1   @T1X_TEMPLATE(IUSHR)
2   public static int iushr(@Slot(1) int value1,
3             @Slot(0) int value2) {
4       return value1 >>> value2;
5   }
6
7   @T1X_TEMPLATE(IRETURN)
8   @Slot(-1)
9   public static int ireturn(@Slot(0) int value) {
10      return value;
```

```
11      }
12
13      @T1X_TEMPLATE(IRETURN$unlock)
14      @Slot(-1)
15      public static int ireturnUnlock(Reference object,
16                  @Slot(0) int value) {
17          Monitor.noninlineExit(object);
18          return value;
19      }
20
21      @T1X_TEMPLATE(IALOAD)
22      public static int iaload(@Slot(1) Object array,
23                  @Slot(0) int index) {
24          ArrayAccess.checkIndex(array, index);
25          int result = ArrayAccess.getInt(array, index);
26          return result;
27      }
28
29      @T1X_TEMPLATE(IASTORE)
30      public static void iastore(@Slot(2) Object array,
31                  @Slot(1) int index, @Slot(0) int value) {
32          ArrayAccess.checkIndex(array, index);
33          ArrayAccess.setInt(array, index, value);
34      }
```

Interpreter and T1X Baseline Compiler

An interpreter is used in the Hotspot VM to translate non-hotspot code into machine language. The interpreter not only interprets bytecode instructions to machine code but also generates debug information such as current bytecode index [PTHH14].

To the contrary, the Maxine VM does not use an interpreter. However, a template-based baseline compiler, called T1X, resembles the role of an interpreter in Maxine. Although acting as a compiler, T1X seeks the speed of machine code generation rather than optimising the generated code [WHVDV$^+$13] [PTHH14]. Templates for bytecode instructions are used to improve the efficiency of this process. Templates consist of

predefined machine code for each bytecode instruction (see Listing 2.1). Compiling bytecode via T1X simply means copying these pre-built machine code instructions to the code buffer (a buffer used to store generated machine code and then inserted into the output binary file). Then, the machine code in the code buffer will be combined with other machine code that may be compiled by T1X or other compilers.

Due to the simplicity of T1X, it has the ability to be easily extended. For example, if researchers want to test a novel translation of a bytecode instruction, they no longer bother to orchestrate a series of complicated sequences in the optimising compiler. All they need to do is to write a template, or even to replace the current corresponding template, in T1X and then test the compilation result. As illustrated in Listing 2.1 creating or replacing entries in T1X is straightforward because these templates are highly independent (decoupled).

However, if only relying on T1X, Maxine will not be able to obtain satisfactory performance. As a result, optimising compilers are used to generate high-quality machine code. Optimising compilers are discussed in the following two sections.

C1X Compiler

The C1X compiler is a Java variation of the optimising client compiler (C1) which is written in C++ and used in Hotspot VM [PTHH14]. Mainly focused on optimising target code, C1X is designed differently from T1X and is more complex. Many optimisation techniques are exploited, such as dead code elimination, global value numbering and block merging [WHVDV$^+$13] [PTHH14].

Graal Compiler

The Graal compiler was firstly created as another Java port of the C1 compiler. Graal can be used by the Maxine VM in a standalone mode, as the only optimising compiler, or in a mixed mode with C1X, as the fail-safe compiler. The combination of Graal and Maxine is shown in Figure 2.2. Notice that Graal uses different levels of intermediate representation (IR) in different compilation stages.

Most modern JVMs use a combination of an interpreter/baseline compiler and

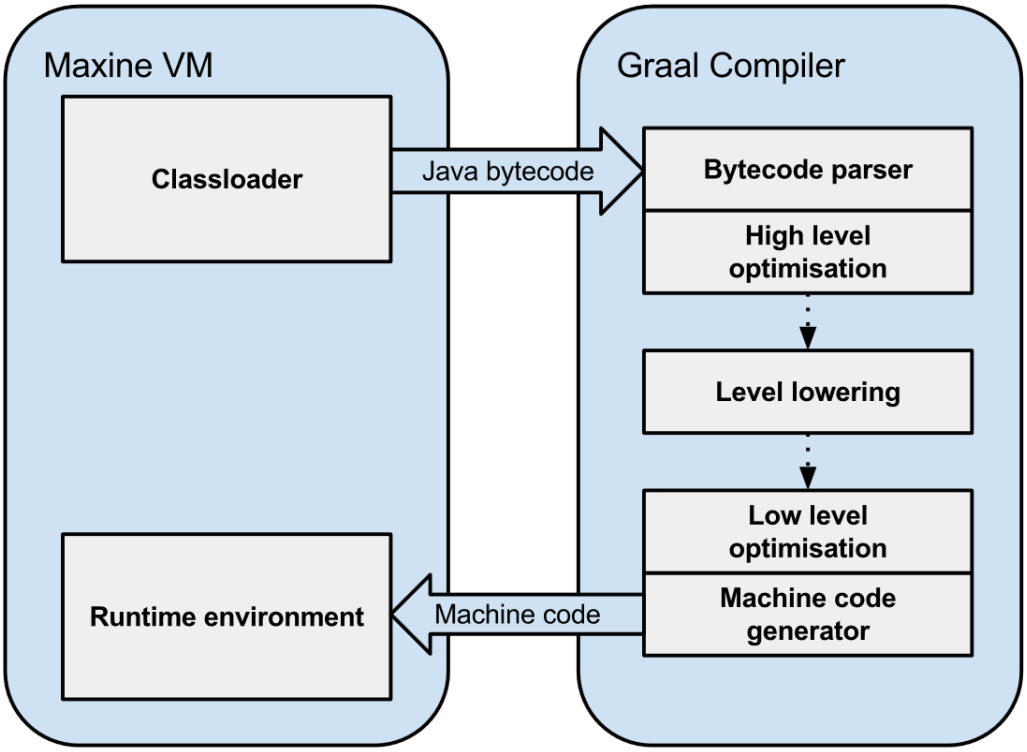

Figure 2.2: Graal reads bytecode, optimises the code in different levels and finally generates target machine code.

an optimising compiler, which is introduced in the beginning of Section 2.2.2. The interaction between interpreters/baseline compilers and optimising compilers are illustrated in Figure 2.3. When a set of bytecode is determined as hotspot code, it will be compiled by the optimising compilers. Optimising compilers perform optimisations to generate machine code with high performance. However, applying these optimisations requires some common hypothesis about the runtime environment and objects [PTHH14]. Sometimes, the compiler may find the hypothesis cannot be satisfied, for example, new classes are loaded or the class hierarchy structure is changed. That is called the uncommon trap [Har14]. In this case, the compiler will perform de-optimising operations within which the bytecode is brought back to the interpreter/baseline compiler. Since interpreters/baseline compilers do not perform optimisations based on hypothesis, they produce stable but relatively verbose machine code, acting like a failsafe procedure.

To summarise, the clearness work flow of T1X and the modular structure of Graal gives good prospect of implementing this project in an incremental approach in which

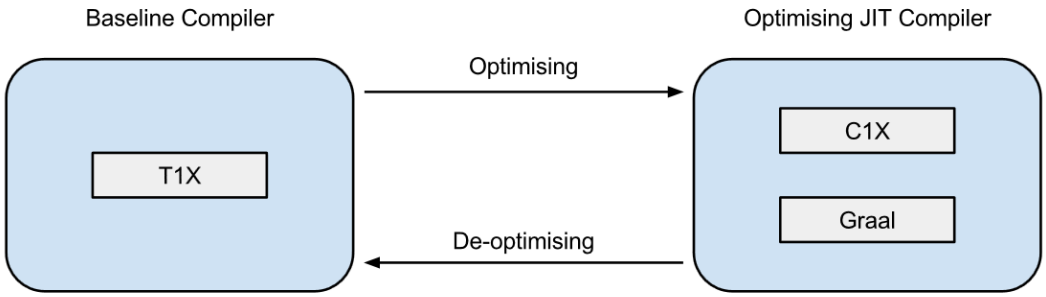

Figure 2.3: Optimising and De-optimising between T1X and C1X/Graal.

each module is implemented independently in an organised manner.

2.3 ARMv8 Architecture

The ARMv8 architecture was introduced in 2011 as the next generation of ARM [Gri11]. This state-of-the-art architecture includes a number of innovations in aspects of high performance computing and energy efficiency.

2.3.1 ARMv8 Overview

Before the ARMv8 architecture is introduced, the following terms are listed to avoid misunderstanding because they are often confusing.

ARMv8

ARMv8 is the name of the 64-bit ARM architecture introduced in 2011.

AArch64

AArch64 is the 64-bit execution state defined by the ARMv8 architecture.

AArch32

AArch32 is the 32-bit execution state defined by the ARMv8 architecture.

A64

A64 is the instruction set used in AArch64 execution state, providing 64-bit

support with fixed-length 32-bit instruction encodings.

A32

A32 is an instruction set used in AArch32 execution state with fixed-length 32-bit instruction encodings.

T32

T32 is another instruction set used in AArch32 execution state with variable-length (16-bit and 32-bit) encodings.

The ARMv8 architecture is an implementation of the Reduced Instruction Set Computer (RISC) architecture. Like most RICS architectures, ARMv8 comes with the following RICS features:

- A large number of registers.
- A load/store architecture where almost all instructions are performed on registers rather than on main memory. Exceptions are load/store instructions which read/save data from/to main memory.
- Simple addressing modes.

Several significant features and improvements are brought by the ARMv8 architecture:

- Registers and addressing space

  The ARMv8 architecture supports 64-bit operation and addressing [Gri11]. Compared to previous 32-bit ARM architectures, such as ARMv7 [ARM12], ARMv8 provides more general-purpose registers (r0 r30) and floating point/SIMD (single instruction, multiple data) registers (v0 v31). In addition, these registers are larger in size than the ARMv7 architecture (64-bit for general-purpose registers and 128-bit for SIMD registers).

- Execution states

  In ARMv8, two execution states are defined: AArch64 and AArch32. They are used to indicate in what size the registers are used and what instruction set is used.

- Instruction sets

  ARMv8 defines three instruction sets, A32, T32 and A64. A32 and T32 are used under AArch32 execution state bringing compatibility with earlier ARM architectures. A64 is a new instruction set only used in AArch64 execution state. As part of a RICS architecture, the A64 instruction sets contains instructions of the same size. Although these instructions are still of 32-bit in size which is the same to ARMv7, their encoding system is changed entirely to fit in the new 64-bit addressing mode.

- Exception model

  The traditional exception model used by previous ARM architectures was originally designed for Acorn computers along with operating system running on it. Nowadays it Acorn computers no seldom used and can only be found in some hobbyists systems. Taking the opportunity of redesigning the whole architecture, ARMv8 is able to adopt a brand new exception model which provide a separation of different privilege levels used by different layers of software.

- Virtualisation

  ARMv8 introduced virtualisation in hardware level. This hardware accelerated virtualisation provide an efficient approach for software logic partitioning and management.

In the next sections, key ARMv8 features that are closely related to the project are discussed in depth.

2.3.2 Instruction Set

As introduced in Section 2.3.1, ARMv8 provides three instruction sets: A64, A32 and T32. In this section, the A64 instruction set is discussed in detail since it is the major instruction set that this project focus on.

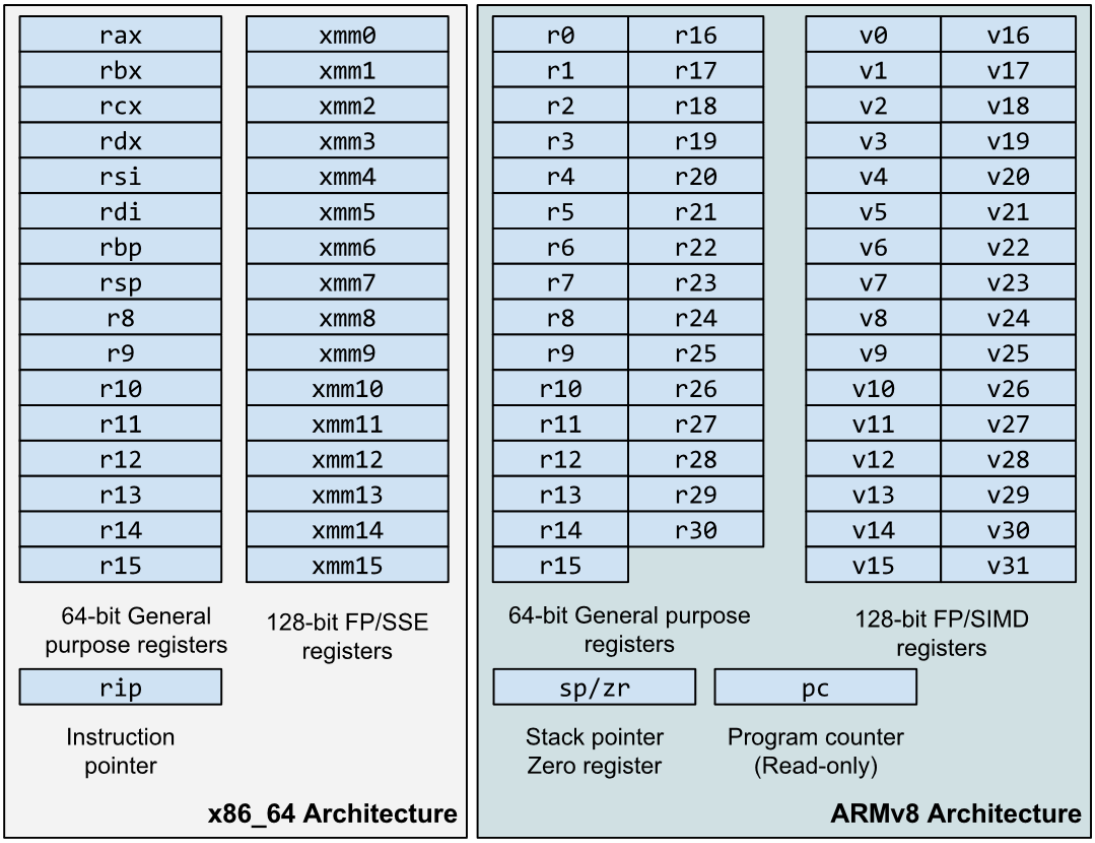

Figure 2.4: Registers in x86_64 and ARMv8.

2.3.3 Registers

Register allocation is a critical aspect when a compiler generates machine code. Since registers are much faster than memory, efficient use of registers can significantly boost the performance of programs. Therefore, in this section the features of ARMv8 registers are studied in order to consolidate the VM/compiler-level implementation of the ARMv8 architecture.

As shown in Figure 2.4, ARMv8 provides more registers that are available to developers compared to the x86_64 architecture. General purpose registers are served as integral registers dedicated to scalar integer computation and memory addressing. In integer computation mode, these registers can be used in 32-bit and 64-bit while in addressing mode they can only be used in 64-bit mode.

One noticeable thing is that the Stack Pointer (SP) can only be accessed in limited situations:

- Load and store instructions

  The stack pointer has to be quad-word aligned (aligned to 16 bytes). If not, a *stack alignment fault* will be issued.

- Add and subtract instructions

  In this case, the stack pointer can be used as the pointer to the location of data.

- Logical data processing instructions

  Same as item 2, the stack pointer is used to indicate the location of source or destination addresses.

In addition to the general purpose registers mentioned above, ARMv8 provides 32 128-bit floating point/SIMD registers (v0 v31). For single and double precision floating computation, they can be accessed in 32-bit or 64-bit (similar to general purpose registers). For single instruction, multiple data instructions, they can be used in 64-bit or 128-bit mode.

2.3.4 Exception Levels

Modern processors use a concept called processor modes or privilege levels to indicate what privilege to place restrictions on how the running code can interact with the processor or its peripherals. In the ARM architecture, application level software has the lowest privilege. As the code being executed, the mode of processor can be changed in two situations:

- The processor will be controlled by privileged code while the code switches the processor to other processor modes;
- An exception occurs and the then processor takes actions to handle the exception.

In the first situation, the change of processor mode is predictable since it follows the sequence of code. In the second situation, the mode is changed automatically by the processor when an exception occurs.

If an exception occurs, the processor will preserve its current state and a return address. The return address tells the exception handler where to return the control after

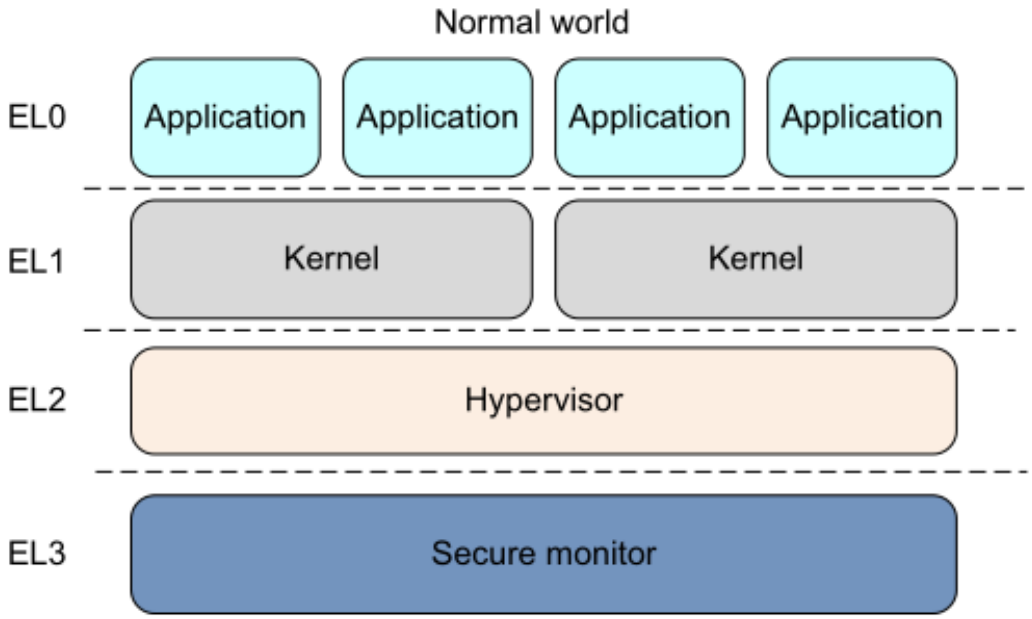

Figure 2.5: ARMv8 Exception Levels.
This figure is taken from Page 3-2 of [ARM15b]

the exception is handled. Then the saved state will be restored to ensure everything is back to normal.

The ARMv8 architecture defines four exception levels EL0-EL3 (see Figure 2.5). They are:

- EL0

  In EL0, the software being executing has the lowest privilege. EL0 is also called the Normal world because most user applications will be executed in this level. In EL0, the processor provides protect against certain software attacks between running processes.

- EL1

  Kernels of operating systems are executed in EL1. The processor provides privileges to the kernel to perform privileged control over user applications in EL0.

- EL2

  In EL2, the processor acts as a hardware hypervisor. A hardware hypervisor is a Virtual Machine Monitor (VMM) [DBR02] built in the architecture, providing

hardware-level virtualisation support. More than one operating systems can be executed under the hypervisor.

- EL3

  EL3 provides a secure monitor to manage different states (secure and non-secure) of the hypervisor.

2.3.5 Application Binary Interface and Stack Layout

The ARMv8 Application Binary Interface (ABI) is the interface lying between the ARMv8 application machine code and operating systems. Mostly, ABI is used by compilers to build binary programs running on a target OS/hardware combination. In other words, compilers have to comply with the ABI to ensure modules that are compiled separately can communicate with each other correctly when they are linked together, thus forming a unified program. In this project, this target combination is GNU Linux on ARMv8 architecture hardware.

Compilers utilise the ABI in several aspects:

- Fundamental data types

  What data types are supported by the target architecture?

- Parameter passing

  How should function/method parameters be passed, using a stack or registers?

- Stack management

  Should the stack be cleaned or created by the callee or caller function?

- Register usage

  How are the registers used in a function call? Which registers are used for parameters and which are used for the return value?

Fundamental data types

The supported fundamental data types are summarised in Table 2.1.

Type Class	Machine Type	Byte Size
Integral	Unsigned byte	1
	Signed byte	1
	Unsigned half-word	2
	Signed half-word	2
	Unsigned word	4
	Signed word	4
	Unsigned double-word	8
	Signed double-word	8
	Unsigned quad-word	16
	Signed quad-word	16
Floating Point	Half precision	2
	Single precision	4
	Double precision	8
	Quad precision	16
Short Vector	64-bit vector	8
	128-bit vector	16
Pointer	Data pointer	8
	Code pointer	8

Table 2.1: Supported fundamental data types in ARMv8

Integral and floating point types are easy to understand since they are quite similar to the primitive data types found in high-level languages like C and Java only with a few extensions like quad precision floating point.

Short vectors are composed of a bunch of elements that are of the same fundamental types, either integral or floating point. For example, a 16-byte short vector can contain 16 signed/unsigned bytes or 4 signed/unsigned words.

Pointers (concretely memory addresses) are 64-bit unsigned data types. Pointers can be used in normal mode or tagged mode. In normal mode, pointer types are identical to the pointers in C/C++, which contain memory addresses. In tagged mode, a tag is integrated into the pointers as the real value of what the pointers point to in memory. This increases the speed of accessing memory through pointers dramatically because the some instructions can read the desired value directly from the tag part of the pointers instead of reading it from the main memory or cache. In a small-memory device, the higher bits of the 64-bit address are always zeros. For example, suppose that a mobile phone using ARMv8 has 2 GB of main memory. The number of bits needed to

address the memory is 31. Then the higher 32 bits can be used to store values which are smaller than 32 bits. This is one of the situations where tagged pointer are useful to deliver performance boost.

Parameter passing

Parameter passing is a critical part of building compilers. High level languages often use complicated data types or structures to represent abstract high level models. This makes it easier for high-level language programmes to write robust programs but requires accurate mappings between these high-level types and the types supported by the hardware.

The process of parameter passing can be defined into the following steps:

1. Mapping original parameter types, which are the types defined by high-level languages, to the fundamental data types described before.
2. Aggregating produced fundamental data type instances to the parameter list.
3. Deciding whether the parameter list should be passed by using registers or a stack in memory.

Register usage

The usage of general-purpose registers are summarised in the following Table 2.2.

Stack management and stack frame

In ARMv8, the stack implementation is full-descending. That is to say , when a *push* operation happens, the stack pointer decreases, which means the stack grows towards the lower address of memory when its size increases. Within the stack, each element must be quad-word aligned (64-bit).

An important concept in Java method calling is the call stack frame [WF98]. Figure 2.6 shows the basic structure of stack frames in a call stack.

Register	Special Name	Usage
SP		Stack pointer
r30	LR	Link register
r29	FP	Frame pointer
r19 ... r28		Callee-saved registers
r18		Platform register
r17	IP1	Second intra-procedure-call scratch register
r16	IP0	First intra-procedure-call scratch register
r9 ... r15		Temporary registers
r8		Indirect result location register
r0 ... r7		Parameter/result registers

Table 2.2: General purpose registers usage

To preserve the context of the caller and callee, a call stack is created to save runtime information. It is usually divided into different but continuous blocks which are named as stack frames. As shown in Figure 2.6, when a caller calls another method (function), a new stack frame is created. Now the stack pointer points to the next free space on the stack top. But a frame pointer (FP) is used to point to the beginning of the current stack frame. The current stack frame contains runtime information of current active method/function. Thus, using the frame pointer, the system can obtain the current running state which is useful for debugging and compilation optimisation.

2.4 Summary

In this chapter, background knowledge of the characteristics and features of the Maxine VM (along with its assembling/compiling components) and the ARMv8 architecture is discussed to the depth of what we need to carry out this project.

To summarise, in order to implement the Maxine VM on the ARMv8 architecture, the following aspects should be and have been studied:

1. Java virtual machine
    a. Role and position a JVM plays.
    b. Structure of the Maxine VM.

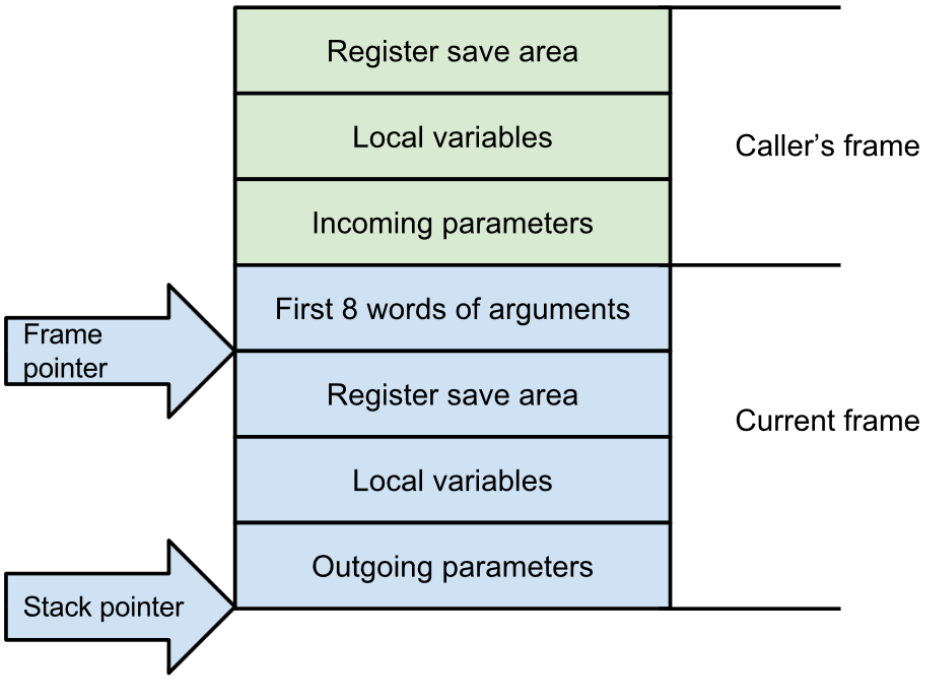

Figure 2.6: ARMv8 stack frame. Frame pointer and stack pointer point to different locations on the call stack.

c. Interaction between assemblers/compilers and the Maxine VM.

2. ARMv8 architecture

a. Instruction sets provided by ARMv8 and used by the Maxine assembler in this project.

b. Application binary interface, including registers functionalities and the stack layout, used by the Maxine assembler and T1X/Graal compilers.

Chapter 3

Design and Implementation

3.1 Implementation Overview

As introduced in Section 2.2 and 2.2.2, the major parts of a fully working Maxine VM implementation on the ARMv8 architecture can be organised into the following modules:

- Architecture Representation (see Section 3.4).
- Assembler Implementation (see Section 3.5).

  ARMv8 Assembler.

  ARMv8 Macro Assembler.
- Compiler Implementation.
- Boot-loader / Substrate.

The first two modules are implemented in this project.

3.2 Methodology - An Incremental Approach

In this section, the methodology and procedures of achieving the items listed in Section 3.1 are discussed. The overall model of how the work-flow is scheduled is illustrated in Figure 3.1.

Because Maxine VM is well defined in a modular scheme, an incremental approach can be used to perform its implementation on ARMv8. In this approach, the implementation of each module is focused on one single topic at each stage with fewest other modules involved. Thus the implementation is clearer and less error-prone.

This incremental approach is described as follows:

- Analysing the x86_64 representation in the Maxine VM.
- Implementing a prototype ARMv8 representation according to the x86_64 representation.
- Tracking the initialisation stage of the Maxine VM to obtain required configurations of x86_64 initialisation.
- Creating ARMv8 configurations according to the ones found in the step above.
- Implementing ARMv8 Maxine assembler system which includes an assembler and a macro assembler.

The existing x86_64 architecture support is used as a reference to the ARMv8 implementation. By tracking and debugging the x86_64 implementation, it is easier to perform the design and implementation for ARMv8.

The above items can be used as a detailed plan for the project. Each item is regarded as a sub-goal which can be fitted into the overall work model shown in Figure 3.1.

3.3 Development Environment

This section describes the setup of the environment needed to research the implementation of Maxine VM on ARMv8. To carry out this project, the host platform and

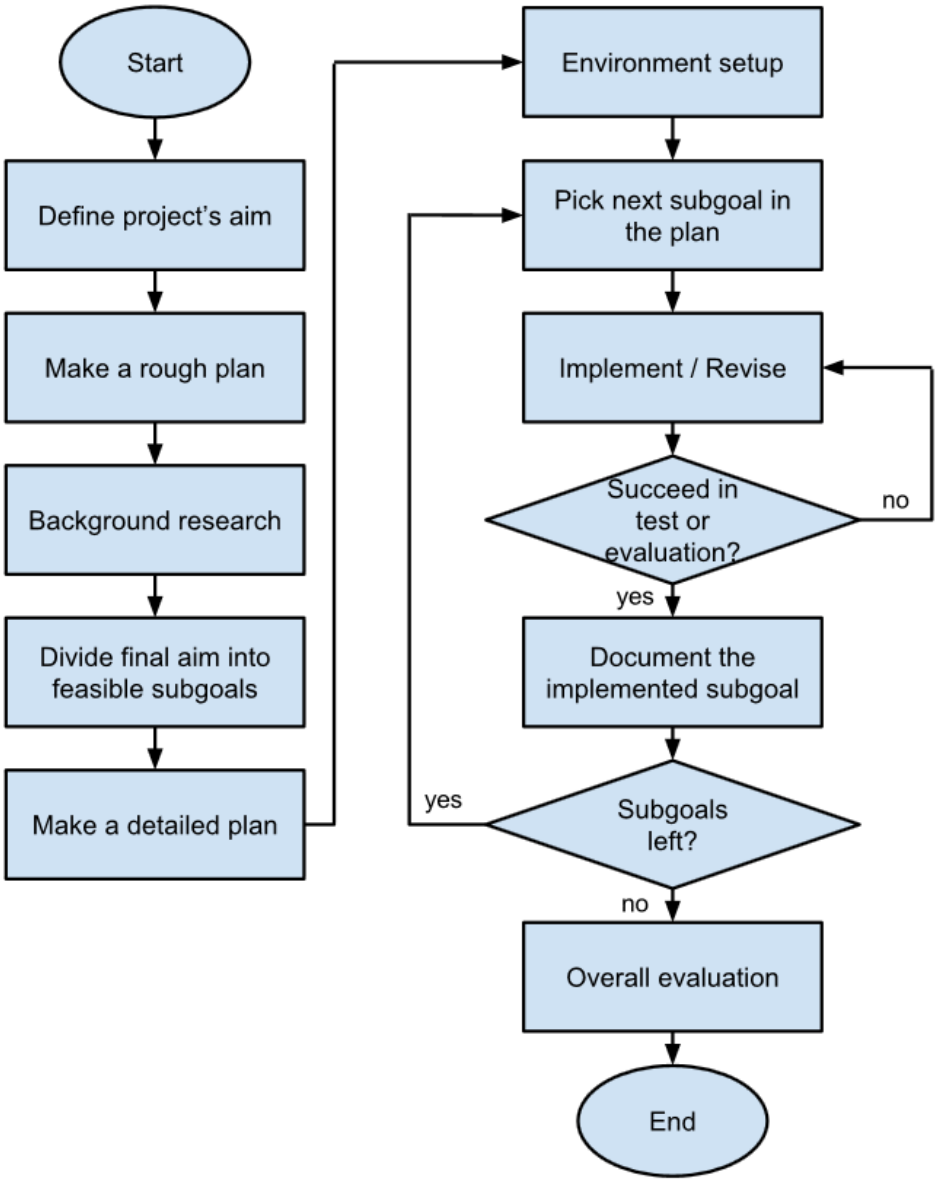

Figure 3.1: Overall research method structure.

tool chain are shown as followings:

- Host operating system

  64-bit GNU/Linux is needed to build the Maxine VM from source. Long-term support (LTS) version of Ubuntu is selected to ensure the stability of the host platform. In this project, Ubuntu 14.04 x86_64 LTS is used.

- Java Development Kit (JDK)

  Maxine is built with the official Oracle JDK (version 7). Due to some minor but critical differences between Open JDK and Oracle JDK, Maxine is not fully compatible with Open JDK.

- GNU Compiler Collection for ARMv8 (Linaro GCC) [Lin14]

  Linaro GCC is used to build ARMv8 binary files.

- Eclipse

  Eclipse is used to build most of the Maxine VM Java code. An integrated development environment (IDE), such as Eclipse, is useful especially when handling a complicated software system. Source code of the Maxine VM is organised into many projects which can be imported to Eclipse.

- ARMv8 Emulator (based on QEMU)

  QEMU is an open source machine emulator with recently added ARMv8 emulation support. The generated machine code is tested on QEMU to verify the correctness of the code.

3.4 ARMv8 Architecture Representation

The architecture/platform initialisation flow of the Maxine VM can be briefly described as follows:

1. The start point of the Maxine VM Java world is the `MaxineVM` class. It loads an instance of the `Platform` class to determine the target architecture.

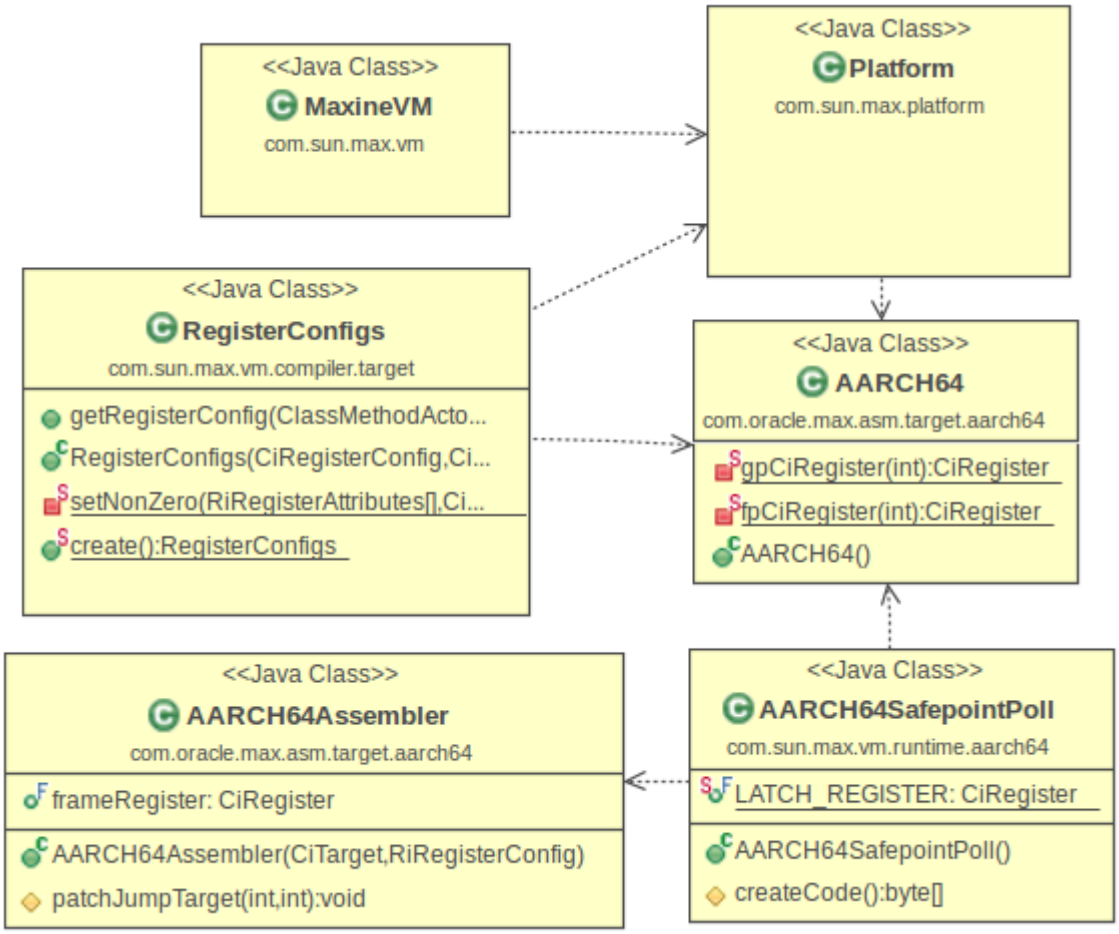

Figure 3.2: An overview UML class diagram of essential ARMv8 representation classes.

2. Within the `Platform` class, target architecture definitions, such as `AARCH64` (for ARMv8), are loaded.

3. The target architecture definitions integrate register configurations, assembler profiles and other architecture characteristic, such as *safe points* (JVM pause points in which actions, such as garbage collection, can be triggered without hazard).

Figure 3.2 illustrates the process in a simplified UML chart where only representative fields/methods are shown.

The `MaxineVM` class is the entry class of the virtual machine. As can be seen from the dashed arrow lines (which means dependency in UML) in Figure 3.2, `MaxineVM` requires platform definitions such as AMD64 (x86_64) and AARCH64 (ARMv8). The

`AARCH64` class, which will be discussed further in the next section, is used to define the register structure and platform features of ARMv8, such as *memory endianness* (byte order) and *memory barriers*.

The `RegisterConfigs` class holds the configuration for each platform and is needed to define register allocation rules, such as *caller/callee register layout* and *parameter passing layout*. It reads platform specific register behaviour and layout requirement to generate correct configurations. `AARCH64SafepointPoll` reads an instance of `AARCH64` to obtain the register layout and then identifies *safe points*.

3.4.1 Registers Representation

Register representation, written in the `AARCH64` class, is the central part of the ARMv8 architecture representation. In this project, all later work is based on how registers are represented and organised. Registers are grouped into three categories because in different contexts they are used for specific purposes. For example, the integer registers are mainly used to pass parameters, convert addresses and perform logic/arithmetic calculations while floating point/SIMD registers are targeted at floating point operations and even cryptography computations. The definition of the `CiRegister` class constructor is shown in Listing 3.1. The property of a register consists of the following items:

1. The number of the register in the register list. (for the register list, please refer to Figure 2.4 and Listing 3.2)
2. The encoding used to index it.
3. The slot size of the registe.
4. The human-readable name of the register.
5. The flag used to identify the type of the register, such as integer or floating point.

Concisely explained, `spillSlotSize` is the size of a register. The reason this parameter is needed is that when a compiler allocates variables in a computer system, it has to decide which should be allocated to registers and which to the memory. The

action of moving one or more variables from register to memory is called *register spilling*. In this case, `spillSlotSize` is needed to determine how much memory space should be allocated. Since different kinds of registers have different sizes, the `flag` parameter is required to identify whether they are integer registers (*CPU_REGISTERS*) or floating point registers (*FPU_REGISTERS*).

Listing 3.1: Constructor of the `CiRegister` class.

```
1
2    public CiRegister(int number, int encoding, int spillSlotSize,
3           String name, RegisterFlag... flags) {
4        this.number = number;
5        this.name = name;
6        this.spillSlotSize = spillSlotSize;
7        this.flags = createMask(flags);
8        this.encoding = encoding;
9
10       values = new CiRegisterValue[CiKind.VALUES.length];
11       for (CiKind kind : CiKind.VALUES) {
12           values[kind.ordinal()] = new CiRegisterValue(kind, this);
13       }
14   }
```

Listing 3.2: ARMv8 register groups.

```
1
2     public static final CiRegister[] cpuRegisters = {
3         r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13,
4         r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24,
5         r25, r26, r27, r28, r29, r30, r31, sp, zr
6     };
7
8     public static final CiRegister[] fpuRegisters = {
9         d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13,
10        d14, d15, d16, d17, d18, d19, d20, d21, d22, d23, d24,
11        d25, d26, d27, d28, d29, d30, d31
12    };
13
14    public static final CiRegister[] allRegisters = {
```

```
15          r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13,
16          r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24,
17          r25, r26, r27, r28, r29, r30, r31, sp, zr,
18          d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13,
19          d14, d15, d16, d17, d18, d19, d20, d21, d22, d23, d24,
20          d25, d26, d27, d28, d29, d30, d31
21      };
22
23      public static final CiRegister[] calleeSavedRegisters = {
24          r19, r20, r21, r22, r23, r24, r25, r26, r27, r28
25      };
```

Each of the registers, such as r0, r1, d0, d1 and so on, in Listing 3.2 are instances of `CiRegister` These registers are grouped into *CPU (integer)* registers, *FP (floating point)* registers and *ALL* registers. In addition the `calleeSavedRegisters` array holds the registers that should be saved by the callee function/method.

The `CPU_REGISTERS` array is consisted of integer registers, which not only contains general purpose registers (r0 ... r30) but also stack pointer (SP), zero register (ZR) and a dummy place-holder r31 which is used to ensure that the encoding is continuous.

In ARMv8, SP and ZR cannot be used as general purpose registers. The stack pointer has been introduced in Section 2.3.3. The zero register is used to get *zero* when needed. SP and ZR are the same register but are accessed via different ways according to specific instructions. When the instruction indicates that SP should be used, ARMv8 processor will provide the real content to the instruction. When the instruction wants to generate zero using ZR, the processor will hide the content and return zero to the instruction. The encoding system ensures that there is no ambiguity of the usage of SP and ZR.

Notice that register r31 is a place-holder rather than a real general purpose register. It represents either SP or ZR depending on the instruction. The reason it is contained in the register list array is that the numbering and encoding of the registers must be sequential, which means that we cannot skip the 32th register (r31) because the arrays need to calculate the index of all registers using their sequential numbers. In the register allocation algorithm, r31 will never be used explicitly.

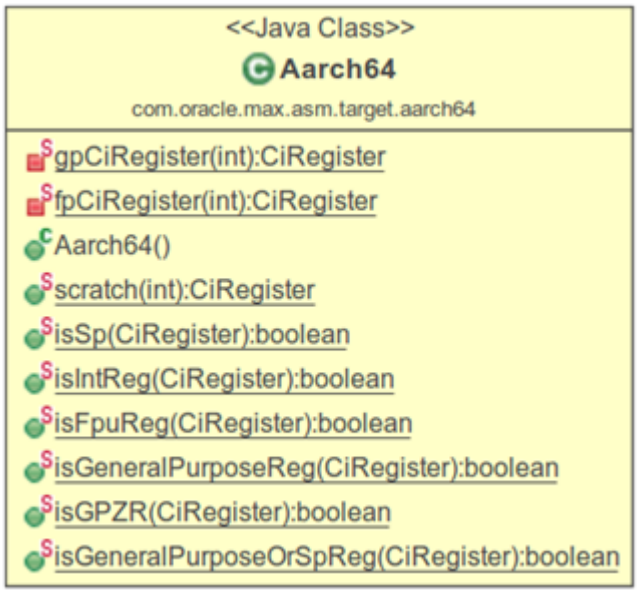

Figure 3.3: Constructor and register verification methods.

The `FPU_REGISTERS` array is used to maintain floating point/SIMD registers (v0 ... v31). These registers are used to perform large data computation or multimedia operation.

The `allRegisters` array is a simple combination of the above two arrays which is used to give the `Platform` class information to create the global register table.

The basic functionalities of the ARMv8 architecture class (`Aarch64`) are shown in Figure 3.3. The bottom half of the UML class diagram lists the verification methods used to check whether a give register, such as `r1`, is a floating point register, an integer register or even other system registers. These methods are useful since it is important to ensure registers are used in the desired manner. Otherwise fatal faults, such as a protection shut down or unhandled exception loops, would happen.

3.4.2 Architecture Configuration Initialisation

After the basic architecture representation class (`Aarch64`) has been implemented, the architecture, sometimes also called *platform* configuration initialisation can read the required information from the `Aarch64` class and set up the basic configuration profile for the ARMv8 assembler and the T1X and Graal compilers.

The initialisation code fragment is listed in Listing 3.3:

Listing 3.3: Code fragment of ARMv8 platform initialisation.

```
1   private CiTarget createTarget() {
2       CiArchitecture arch = null;
3       int stackAlignment = -1;
4       if (isa == ISA.AMD64) {
5           ...
6           ...
7       } else if (isa == ISA.Aarch64) {
8           arch = new Aarch64();
9           if (os == OS.LINUX) {
10              stackAlignment = 16;
11          } else {
12              throw FatalError.unexpected(
13                  "Unimplemented stack alignment: " + os);
14          }
15      }
16      else {
17          return null;
18      }
19
20      assert arch.wordSize == dataModel.wordWidth.numberOfBytes;
21      boolean isMP = true;
22      int spillSlotSize = arch.wordSize;
23      int cacheAlignment = dataModel.cacheAlignment;
24      boolean inlineObjects = false;
25
26      return new CiTarget(
27              arch,
28              isMP,
29              spillSlotSize,
30              stackAlignment,
31              pageSize,
32              cacheAlignment,
33              inlineObjects,
34              false,
35              false);
```

```
36      }
```

The initialisation first checks the target operating system and processor architecture combination, in this case *Linux + Aarch64*, and fill the necessary platform parameters, which are obtained from the architecture combination, in the constructor of the new Target class (`CiTarget`). `CiTarget` is then used to offer such information to the ARMv8 Maxine assembler system and the T1X and Graal compilers.

3.5 ARMv8 Assembler and Macro Assembler

After the ARMv8 architecture representation is complete, the assembler system for ARMv8 is the next implementation stage.

The role of the ARMv8 assembler can be illustrated in Figure 3.4. The Maxine Assembler System (MAS) for ARMv8 consists of an assembler and a macro assembler. The former assembles (translates) each ARMv8 assembly instruction into a ARMv8 machine code instruction binary sequence (32-bit). The later translate a macro (a sequence of several instructions) into pre-defined machine code instructions.

The relationship between the architecture representation discussed in Section 3.4 and the MAS is close because the ARMv8 assembler uses the architecture information, such as register indexes and the stack convention, to assemble instructions into machine code.

To make the implementation of the assembler easy to extend and clear to understand, an enumeration-based structure is used. This structure is illustrated in Figure 3.5. The fundamental idea is that due to the encoding simplicity of ARMv8 instruction set, which is discussed in Section 2.3.2, instruction encodings are organised by several levels of enumerations (implemented as Java `enum` types). When generating the machine code, firstly the assembler determines each part of the instruction and read the encoding from corresponding enumerations.

Details of this *enumeration-based* structure are introduced in the following sections, along with the major components of the assembler and the macro assembler explained.

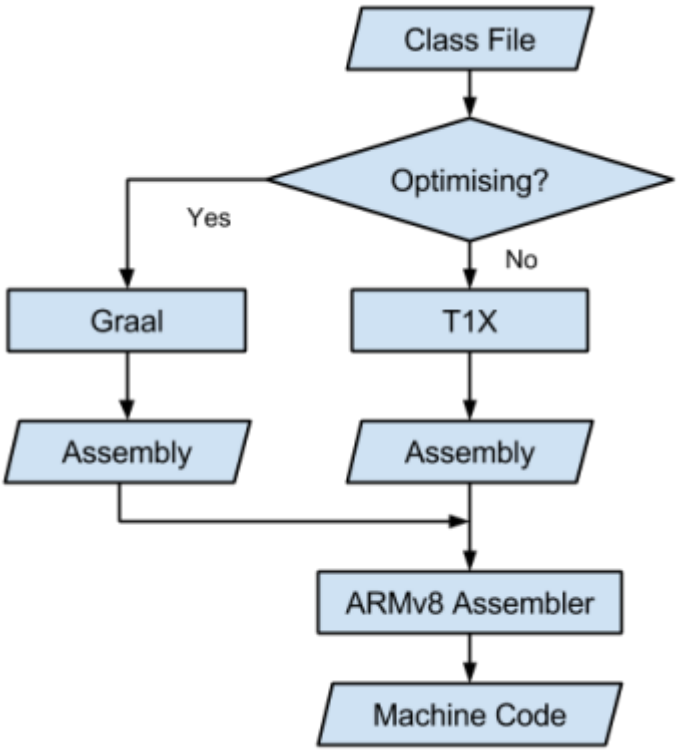

Figure 3.4: The role of the ARMv8 assembler in the Maxine VM.

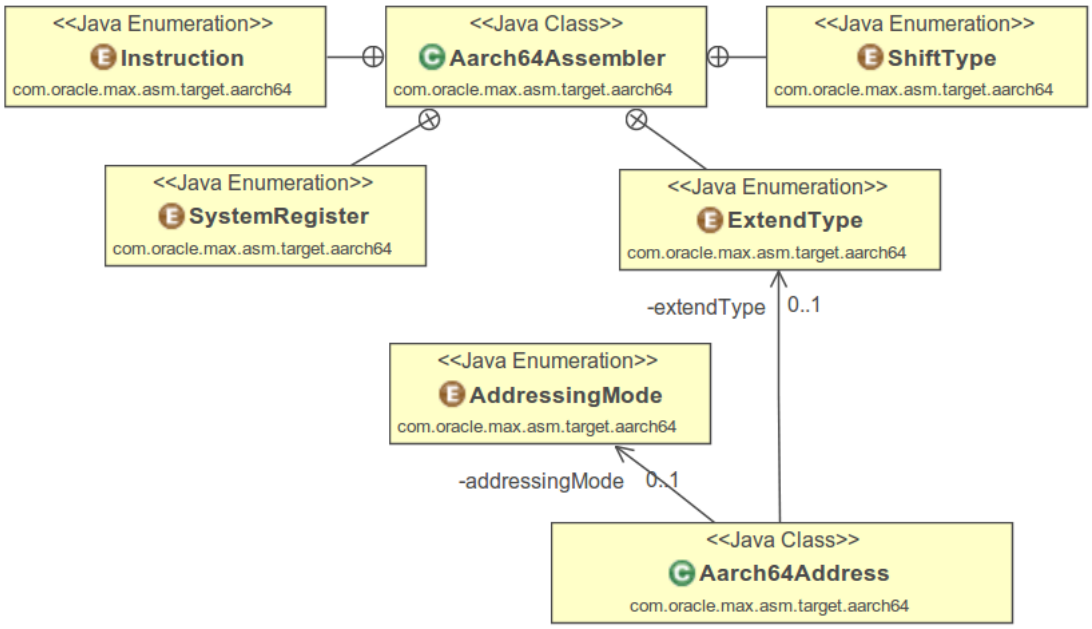

Figure 3.5: An enumeration-based structure of the ARMv8 assembler.

3.5.1 Instruction Encoding

Instruction encoding is the central component of the ARMv8 assembler.

To explain the encoding system better, the encoding rules for addition and subtraction instructions are listed in Figure 3.6 and Table 3.1 as examples.

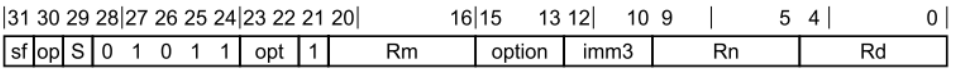

Figure 3.6: Bit-wise encoding for addition and subtraction instructions (extended register type).

Decode fields						
sf	op	S	opt	imm3	Instruction	Variant
0	0	0	00	-	ADD (extended register)	32-bit
0	0	1	00	-	ADDS (extended register)	32-bit
0	1	0	00	-	SUB (extended register)	32-bit
1	1	1	00	-	SUBS (extended register)	32-bit
1	0	0	00	-	ADD (extended register)	64-bit
1	0	1	00	-	ADDS (extended register)	64-bit
1	1	0	00	-	SUB (extended register)	64-bit
1	1	1	00	-	SUBS (extended register)	64-bit

Table 3.1: Encoding table for addition and subtraction instructions (extended register type).

As illustrated in Figure 3.6 and Table 3.1, addition and subtraction instructions are encoded under a unified rule in which several bit fields are used to identify all instruction variants, such as ADD, ADDS, SUB and SUBS, that are related to each other. For example, the `sf` (*sixty-four*) bit field indicates if this instruction is a 64-bit variant and the `op` (*operation*) bit field shows whether this instruction is addition or subtraction.

The basic idea of implementing this encoding scheme is to emit the binary machine code progressively in separated steps. Figure 3.7 shows the process of emitting the machine code of the `ADD X0, X1, #0x3` instruction. The encoding bit fields of this instruction consists of the following:

1. Instruction Base Encoding.
2. Variants.
3. Destination Register.
4. Source Register.
5. Immediate Number.

The final instruction encoding for `ADD X0, X1, #0x3` is the exclusive or (XOR) result of the five bit fields listed above.

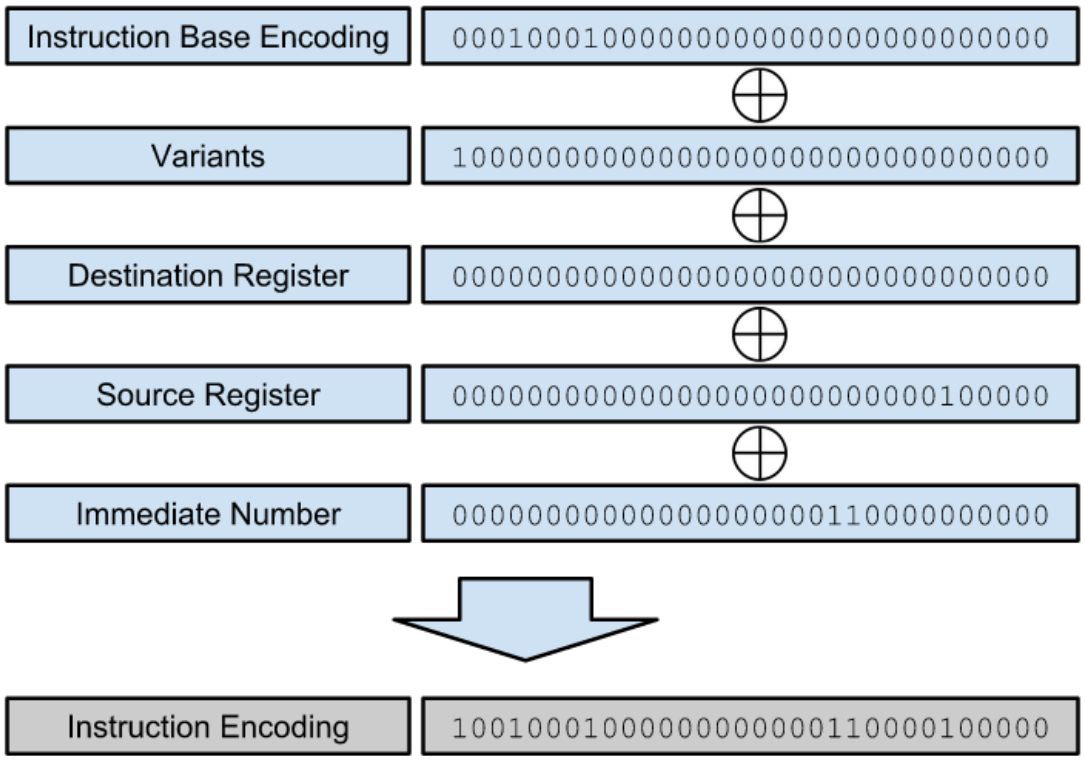

Figure 3.7: Steps of emitting the binary machine code of the `ADD X0, X1, #0x3` instruction.

The code shown in Listing 3.4 is an implementation of the process described above Please be aware that the code listed here has been rewritten to improve its readability. As a result it is verbose to some extent. Code from Line 6 to 10 reads the integer value of the encoding for each bit field. Code from Line 12 to 17 integrate the bit fields into a complete instruction. In Line 19, the `emit()` method inserts the finished instruction binary encoding into a machine code buffer.

Listing 3.4: An example of generating ADD/SUB instructions binary code

```
1   void add(
2           CiRegister dst, CiRegister src, int aimm,
3           InstructionType type, Instruction instr) {
4           int instrEncoding = instr.enc | AddSubImmOp;
5
6       int type_enc = type.encoding;
7       int instr_enc = instrEncoding;
8       int imm_enc = encodeAimm(aimm);
9       int rd_enc = rd(dst);
10      int rs_enc = rs(src);
11
12      int enc =
13          type_enc |
14          instr_enc |
15          imm_enc |
16          rd_enc |
17          rs_enc;
18
19      emit(enc);
20  }
```

All `enum` types shown in Figure 3.5 can be used as part of the encoding fields.

The number of system registers in ARMv8 is more than one hundred [Goo11]. It is inefficient to implement them at the first place since most of them are not needed by most programs. But it is still important to keep a good capability of extending the system registers when they are needed later. Thus using an extensible `enum` type is a reasonable choice. Listing 3.5 shows the convenience of applying this `enum`-based encoding system.

Listing 3.5: System register encodings

```
1   /**
2    * This enum is used to indicate how system registers
        are encoded.
3    */
4   public static enum SystemRegister {
5       NZCV   (0b1101101000010000),
6       DAIF   (0b1101101000010001),
7
8       ...
9       ...
10
11      SPSel  (0b1100001000010000),
12      SPSR_EL1(0b1100001000000000);
13
14      public final int encoding;
15
16      private SystemRegister(int encoding) {
17          this.encoding = encoding;
18      }
19  }
```

The code in Listing 3.5 maintains an `enum` type that contains system registers that are currently implemented. Throughout the code base, they are referred to using the `enum` type name `SystemRegister`. Whenever a system register encoding should be implemented, it can be added into the `enum` directly without changing any other part of the code base.

3.5.2 Address Representation

Memory addressing in modern computer architectures can be complicated. ARMv8 provides several load/store addressing modes (see Table 3.2).

Notice that some instructions do not supported every addressing mode due to the limit of the length of bit fields used for address encoding in each instruction. This

Type	Immediate Offset	Register Offset
Simple register (exclusive)	[base{,#0}]	-
Offset	[base{,#imm}]	[base,Xm{,LSL #imm}]
Pre-indexed	[base,#imm]!	-
Post-indexed	[base],#imm	-
PC-relative (literal) load	label	-

Table 3.2: Load/store addressing modes in ARMv8

means that the assembler must be responsible for checking the correctness of the addressing mode used in instructions.

To explain the addressing implementation, this section describes the following items in detail:

- Detailed ARMv8 addressing modes
- Creating an ARMv8 memory address
- Verifying an address
- Using an address

ARMv8 addressing modes

Addressing modes listed in Table 3.2 are not precise enough to create a general address representation in this project. The assembler specifies more addressing modes that are derived from the above ones:

- Base register only

  This is the simplest addressing mode where the specified base register is the only addressing register. An example can be:

  ```
  LDR, X0, X1
  ```

  This instruction loads the memory content *from* the address represented by the value of the `X1` register *into* the destination register `X0`.

- Base register + immediate number

Similar to the previous addressing mode, but the memory address is represented by both a base register and an immediate number. The immediate number is added to the base register value. Below is an example:

```
LDR X0, [X1, #3]
```

- Base register + offset register

  In this mode, the address is represented by both a base register and an offset register, meaning that:

  address = *base* + *offset*.

  The offset register can be used in either unscaled (direct) mode or scaled mode (scaled by the size of each transferred element). The following instruction shows the form of unscaled offset register addressing mode.

  ```
  LDR X0, [X1, X2]
  ```

  The target address is calculated by adding the values of `X1` and `X2` registers. Then the content of the memory address is loaded into the `X0` register.

- Program counter (PC register) + label / literal value

  Two examples can be:

  ```
  LDR X0, LABEL_1
  ```

  ```
  LDR X0, =1234
  ```

  The first instruction loads the value of the LABEL_1 label into the `X0` register. The second one loads the literal value `1234` into `X0`.

  The assembler handles this addressing mode by converting the label or literal value into addresses that point to either where the label is or the literal pool where literals are stored. Thus the converted version of the instruction looks like the following:

  ```
  LDR X0, 0x40081008
  ```

  The converted format is not intended for human to read but for the machine to execute. Programmers using this assembler can write instructions in human-readable format, such as the above two instructions, and let this assembler to convert it to the compact version.

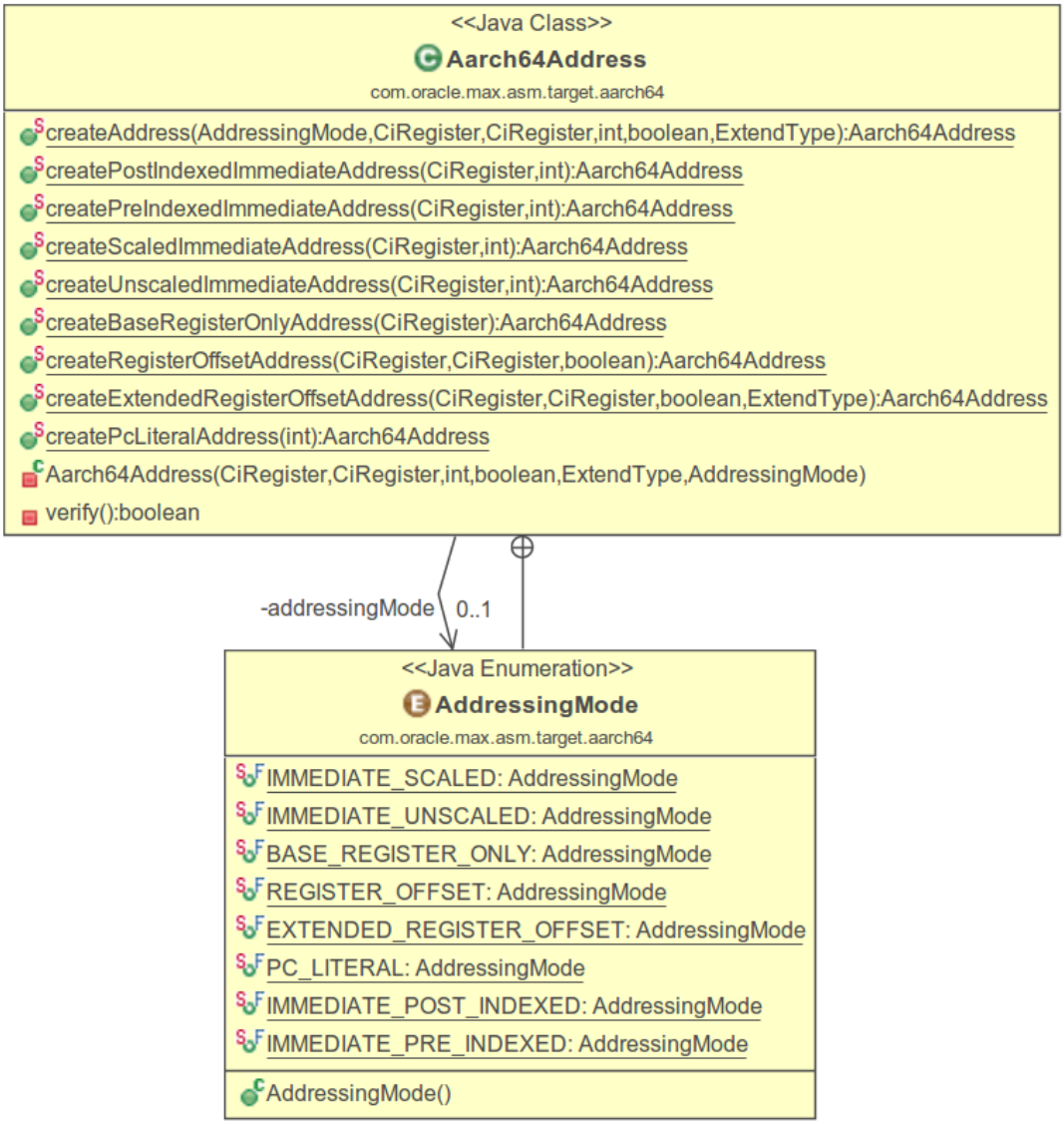

Figure 3.8: A simplified UML class diagram for `Aarch64Address` along with its addressing modes enumeration.

Address creation

In the ARMv8 assembler, integer load (LDR) and store (STR) instructions are organised in a unified manner respectively. That is to say, although they can be used in different addressing modes, their encoding process is the same. This is done by introducing an `Aarch64Address` class that handles the various addressing mode in a clear way.

Figure 3.8 describes the major behaviours of the `Aarch64Address` class.

`Aarch64Address` handles different addressing modes using the *factory method* pattern [GHJV94]. Factory methods, starting with *create*, identify different modes and call the class constructor with corresponding parameters.

As a result, the encoding process of LDR and STR instructions no longer cares about the different addressing modes but only provides proper parameters to the factory methods. This makes the assembler system extensible and easy to incorporate additional features.

3.5.3 ARMv8 Macro Assembler

The need for a macro assembler comes from the fact that there are some functionalities used very often by most programs. Each of these functionalities is consisted of several fixed ARMv8 instructions. Some examples are:

- Generating a memory address from given registers.
- Loading a memory address into a register.
- Saving registers to memory to preserve execution context.

Since they are frequently used, it is convenient to implement them using predefined macros, which makes the assembling process clearer without having to write the code manually and repeatedly.

The macro assembler also provides *pseudo instructions* translation. The T1X and Graal compilers require some macros because they produce *pseudo instructions* that

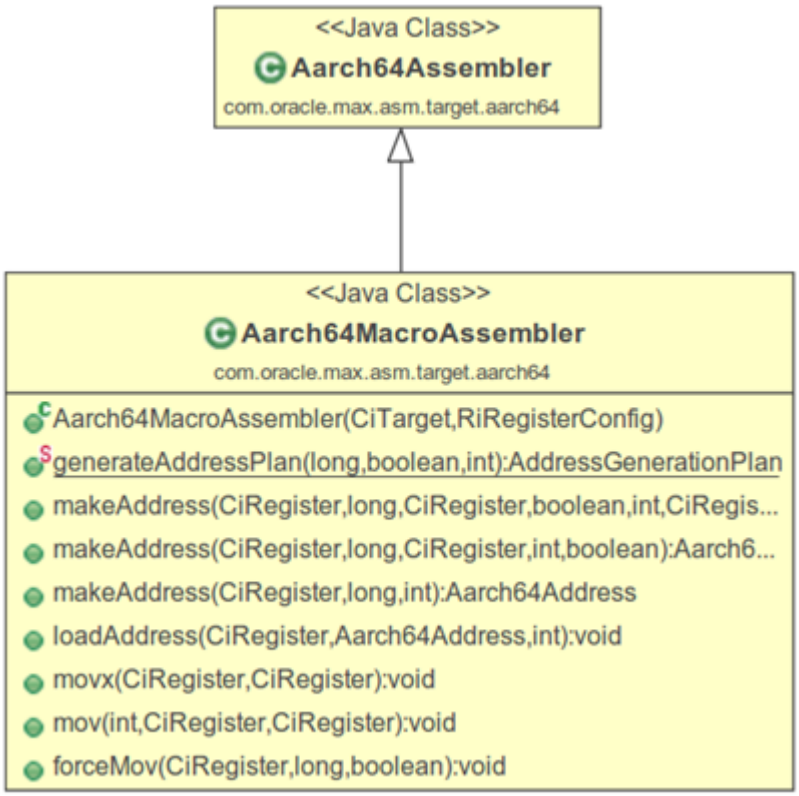

Figure 3.9: A simplified UML class diagram for `Aarch64MacroAssembler` and its parent class `Aarch64Assembler`.

are not standard ARMv8 instructions. Some of these pseudo instructions are processed by the ARMv8 macro assembler.

Figure 3.9 is a simplified UML class diagram. Some, not all, pre-built macros are listed. For example, the `makeAddress()` methods are used to create ARMv8 addresses from the information given by the parameters and the `mov()` methods offer a simpler way to transfer data between registers.

3.6 Summary

In this section, the methodology and the development environment used to carry out the project have been introduced. In addition, the design and implementation of the following components of Maxine VM are discussed as well:

- ARMv8 architecture representation.
- ARMv8 Maxine assembler system, including the ARMv8 assembler and macro assembler.

The source code of this project is hosted at repositories `https://bitbucket.org/nisbeta-Manchester/maxine-aarch64` and `https://bitbucket.org/nisbeta-Manchester/graal-aarch64`. Permissions can be obtained by contacting Dr. Andy Nisbet (Andy.Nisbet@manchester.ac.uk).

Chapter 4

Result and Evaluation

This chapter introduces the achievement of this project and provides evaluations of the result.

4.1 Achievement

This project has implemented the following components of the Maxine VM:

- ARMv8 architecture representation.
- Maxine Assembler System (MAS) for ARMv8, including an assembler and a macro assembler.

The implemented instructions are listed in Appendix.

4.2 Evaluation

The evaluation of the Maxine assembler system is done by executing the generated ARMv8 binary code on the QEMU ARMv8 emulator and verifying the correctness of the execution result.

4.2.1 Unit Test Framework for ARMv8 MAS

Throughout the evaluation, a unit test framework is used to:

1. Generate an executable binary file from the emitted machine code for the target architecture, in this case, ARMv8.
2. Bridge and exchange data and configuration between Maxine VM and the QEMU ARMv8 emulator.
3. Obtain architecture-level information, such as register contents, from the QEMU emulator.
4. Verify the execution result and compare it with the expected result.

Listing 4.1 shows the `generateAndTest()` method in the `Aarch64AssemblerTest` class. This method generates an executable binary file for ARMv8 hardware, in this case, the QEMU ARMv8 emulator, and then starts the verification of the obtained result. Lines 4 and 5 read the stored instruction machine code buffer from `codeBuffer` and then create an executable binary file. Lines 6 to 14 prepare the runtime environment for the unit-test framework. After the preparation is done, the unit-test framework starts the QEMU ARMv8 emulator with the given information, such as a copy of the binary file, as is shown in line 15.

Listing 4.1: Generating an executable binary file and verify the obtained result.

```
1   private void generateAndTest(long[] expected, boolean[] tests,
2             MaxineARMTester.BitsFlag[] masks, Buffer codeBuffer)
3             throws Exception {
4       ARMCodeWriter code = new ARMCodeWriter(codeBuffer);
5       code.createCodeFile();
6       MaxineARMTester r = new MaxineARMTester(expected, tests, masks);
7       if (!MaxineARMTester.ENABLE_SIMULATOR) {
8           System.out.println("Code Generation is disabled!");
9           return;
10      }
11      r.assembleStartup();
12      r.compile();
```

```
13          r.link();
14          r.objcopy();
15          r.runSimulation();
16          r.reset();
17      }
```

In addition, the `Aarch64AssemblerTest` class provides unit test cases that cover all instructions implemented in the assembler class (`Aarch64Assembler`) introduced in Section 3.5. One example of the unit test case for `LDR` and `STR` instructions are listed in Listing 4.2. Lines 6 to 12 prepare a value (in this case, a hexadecimal integer `0x123`) to *load* and *store*, an address offset and a final stack address based on the stack pointer (`Aarch64.sp`) and the offset. Then with the information prepared, lines 14 to 16 generate the code buffer containing `STR` and `LDR` instructions. The former *stores* the value `0x123` to the desired stack slot pointed by the `address` variable. The later then reads the value from the same stack slot to register `Aarch64.cpuRegisters[1]` or `Aarch64.r1` in short.

If the assembler for `LDR` and `STR` is correct, the final content of `Aarch64.r1` should be `0x123`, which is exactly the value pushed into the stack earlier. To verify this, lines 19 to 20 specify the expected value and which register should be verified. In this case, register `Aarch64.r1` should be verified and the expected value is `0x123`. After all things are set, in line 22 the unit test case calls the `generateAndTest()` method described earlier in this section. Then the verification result will be reported after the execution on the QEMU ARMv8 emulator.

Listing 4.2: Unit test case for `LDR` and `STR` instructions.

```
1   public void test_ldr_str() throws Exception {
2       // initialisation code
3       ...
4
5       // value to be stored to stack
6       asm.movz(VARIANT_64, Aarch64.cpuRegisters[0], 0x123, 0);
7       // stack offset
8       asm.movz(VARIANT_64, Aarch64.cpuRegisters[10], 8, 0);
9       // final stack address
10      Aarch64Address address = Aarch64Address.
```

```
11                  createRegisterOffsetAddress(Aarch64.sp,
12                  Aarch64.cpuRegisters[10], false);
13          // store value to stack
14          asm.str(VARIANT_64, Aarch64.cpuRegisters[0], address);
15          // load value from stack
16          asm.ldr(VARIANT_64, Aarch64.cpuRegisters[1], address);
17
18          // set the expected value and testing flag
19          expectedValues[1] = 0x123;
20          testValues[1] = true;
21
22          generateAndTest(expectedValues, testValues, bitmasks,
23                  asm.codeBuffer);
24      }
```

A unit test result output example is listed in Listing 4.3. As can be seen from the output, register `R0` is not tested since its testing flag is `false`. Register `R1` is tested and its simulation value (`sim:291`, `0x123` in decimal) is consistent with the expected value (`exp:291`).

Listing 4.3: Example of the unit test verification output.

```
1   running test_ldr_str(test.arm.asm.Aarch64AssemblerTest)
2   ...
3   MaxineARMTester:: qemu ready
4   0 sim: 291 exp: 0 test: false
5   1 sim: 291 exp: 291 test: true
6   ...
```

4.2.2 Debug Tool Chain and Disassembly Evaluation

To examine the assembly code at QEMU emulator runtime, the gdb (GNU debugger) command `aarch64-none-elf-gdb` for ARMv8 (AArch64) can be used. Listing 4.4 is the disassembly result captured from `aarch64-none-elf-gdb` output. Note that 64-bit integer registers are shown as `x0 ... x30` instead of `r0 ... r30` in the debugger. The disassembly output is a useful evaluation tool to see if the instruction

encoding works as expected because if not, the disassembly output will give either *undefined* instructions or other instructions that are not expected.

Listing 4.4: Disassembly output for the `LDR/STR` unit test case.

```
1  Dump of assembler code from 0x40081000 to 0x40081100:
2  => 0x0000000040081000: mov x0, #0x123   // #291
3     0x0000000040081004: mov x10, #0x8    // #8
4     0x0000000040081008: str x0, [sp,x10]
5     0x000000004008100c: ldr x1, [sp,x10]
6     0x0000000040081010: b 0x40081010
```

4.3 Summary

Instructions implemented in the assembler are listed in Table A.1 in Appendix A. These instructions are evaluated both by the unit test cases and disassembly output and have generated correct results.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

As the title suggests, the aim of this project is to investigate the implementation of Maxine VM on ARMv8. In Chapter 1, the motivation of this project is described and the basic objectives and deliverables are expressed. In Chapter 2, the required background knowledge and research context are introduced to give clues of how this project are related to the Java technology, Maxine VM and the state-of-the-art ARMv8 processor architecture. Chapter 3 describes the design and implementation of two key modules, the ARMv8 architecture representation and the Maxine ARMv8 assembler system, of Maxine VM on ARMv8. Chapter 4 provides information of how the achievements are evaluated using a tool chain consisting of a unit test framework, the QEMU ARMv8 emulator and the GNU debugger for ARMv8.

The well-defined modular structure of Maxine VM (see Section 2.2) has offered considerable convenience throughout the project. Since each module of Maxine VM can be extended for specific architecture / platform independently, the work load of developing and debugging is reduced.

5.2 Limitations

Due to the limited time of performing this project, although the architecture representation and the Maxine ARMv8 assembler are finished, there are still several important components that are not implemented. They are listed below. As introduced in Chapter 2, these items are indispensable to form a fully working Maxine VM on the ARMv8 architecture.

- T1X compiler.
- Graal or C1X compiler.
- Boot-loader / Substrate.

Another limitation is that the system register encoding is not fully finished as introduced in Section 3.5.1. Currently only a few frequently used system registers are included. In consideration of implementing other important aspects as many as possible in such limited time, we decided to leave the extension job at the moment. Fortunately, extending system registers is not difficult but requires some time.

5.3 Suggested Future Work

5.3.1 T1X compilers

T1X provides template-based compilation to compile Java bytecode to assembly code. Because T1X is the fundamental compiler used in Maxine VM, it is strongly suggested that future researchers implement T1X for ARMv8 first. The ARMv8 assembler and macro assembler is ready and can be used in T1X compilation.

5.3.2 Graal / C1X compilers

Graal and C1X are optimising compilers used to generate optimised assembly code. Currently, the Graal compiler is preferred due to its extension-friendly feature. Graal

can also be used with the Truffle framework to provide a Truffle-based runtime. More information about Truffle and Graal can be found in [WW12].

5.3.3 Boot-loader

The boot-loader, also know as *substrate*, is used to boot Maxine VM on host operating systems. This is the only component that are written in C while other components in Maxine VM are written in Java.

5.3.4 Support for A32 and T32 instruction sets

The Maxine ARMv8 assembler system provides support for the A64 instruction set. It is possible to extend the assembler to support A32 and T32 instruction sets if they are needed. Description of A32 and T32 can be found in Section 2.3.2.

Appendix A

Instructions Implemented in the ARMv8 Maxine Assembler

Table A.1 contains the instructions that are implemented in the ARMv8 Maxine assembler. They are divided into 8 categories according to their usage and encodings. Detailed explanation and encoding rules can be found in Chapters C3, C4 and C5 of the Architecture Reference Manual for ARMv8 [ARM15a]. Details of the implementation can be found in class `com.oracle.max.asm.target.aarch64.Aarch64Assembler`.

Table A.1: Implemented ARMv8 Instructions.

Instruction Type	**Instruction**
Control Flow	b(ConditionFlag,int)
	b(ConditionFlag,int,int)
	cbnz(int,CiRegister,int)
	cbnz(int,CiRegister,int,int)
	cbz(int,CiRegister,int)
	cbz(int,CiRegister,int,int)
	b(int)
	b(int,int)
	bl(int)
	blr(CiRegister)
	br(CiRegister)
	ret(CiRegister)

Continuation of Table A.1

Instruction Type	**Instruction**
Memory Access	ldr(int,CiRegister,Aarch64Address)
	ldrs(int,int,CiRegister,Aarch64Address)
	str(int,CiRegister,Aarch64Address)
	ldxr(int,CiRegister,Aarch64Address)
	stxr(int,CiRegister,CiRegister,Aarch64Address)
	ldar(int,CiRegister,Aarch64Address)
	stlr(int,CiRegister,Aarch64Address)
	ldaxr(int,CiRegister,Aarch64Address)
	stlxr(int,CiRegister,CiRegister,Aarch64Address)
	adr(CiRegister,int)
Data Processing (immediate)	add(int,CiRegister,CiRegister,int)
	adds(int,CiRegister,CiRegister,int)
	sub(int,CiRegister,CiRegister,int)
	subs(int,CiRegister,CiRegister,int)
	and(int,CiRegister,CiRegister,long)
	ands(int,CiRegister,CiRegister,long)
	eor(int,CiRegister,CiRegister,long)
	orr(int,CiRegister,CiRegister,long)
	movz(int,CiRegister,int,int)
	movn(int,CiRegister,int,int)
	movk(int,CiRegister,int,int)
	bfm(int,CiRegister,CiRegister,int,int)
	ubfm(int,CiRegister,CiRegister,int,int)
	sbfm(int,CiRegister,CiRegister,int,int)
	extr(int,CiRegister,CiRegister,CiRegister,int)
Data Processing (register)	add(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	adds(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	sub(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	subs(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	add(int,CiRegister,CiRegister,CiRegister,ExtendType,int)
	adds(int,CiRegister,CiRegister,CiRegister,ExtendType,int)
	sub(int,CiRegister,CiRegister,CiRegister,ExtendType,int)
	subs(int,CiRegister,CiRegister,CiRegister,ExtendType,int)

Continuation of Table A.1	
Instruction Type	**Instruction**
	and(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	ands(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	bic(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	bics(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	eon(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	eor(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	orr(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	orn(int,CiRegister,CiRegister,CiRegister,ShiftType,int)
	asr(int,CiRegister,CiRegister,CiRegister)
	lsl(int,CiRegister,CiRegister,CiRegister)
	lsr(int,CiRegister,CiRegister,CiRegister)
	ror(int,CiRegister,CiRegister,CiRegister)
	cls(int,CiRegister,CiRegister)
	clz(int,CiRegister,CiRegister)
	rbit(int,CiRegister,CiRegister)
	rev(int,CiRegister,CiRegister)
	csel(int,CiRegister,CiRegister,CiRegister,ConditionFlag)
	csneg(int,CiRegister,CiRegister,CiRegister,ConditionFlag)
	csinc(int,CiRegister,CiRegister,CiRegister,ConditionFlag)
Integer Multiply / Divide	madd(int,CiRegister,CiRegister,CiRegister,CiRegister)
	msub(int,CiRegister,CiRegister,CiRegister,CiRegister)
	sdiv(int,CiRegister,CiRegister,CiRegister)
	udiv(int,CiRegister,CiRegister,CiRegister)
Floating Point Operation	fldr(int,CiRegister,Aarch64Address)
	fstr(int,CiRegister,Aarch64Address)
	fmov(int,CiRegister,CiRegister)
	fmovFpu2Cpu(int,CiRegister,CiRegister)
	fmovCpu2Fpu(int,CiRegister,CiRegister)
	fmov(int,CiRegister,double)
	fcvt(int,CiRegister,CiRegister)
	fcvtzs(int,int,CiRegister,CiRegister)
	scvtf(int,int,CiRegister,CiRegister)
	frintz(int,CiRegister,CiRegister)

Continuation of Table A.1	
Instruction Type	**Instruction**
	fabs(int,CiRegister,CiRegister)
	fneg(int,CiRegister,CiRegister)
	fsqrt(int,CiRegister,CiRegister)
	fadd(int,CiRegister,CiRegister,CiRegister)
	fsub(int,CiRegister,CiRegister,CiRegister)
	fmul(int,CiRegister,CiRegister,CiRegister)
	fdiv(int,CiRegister,CiRegister,CiRegister)
	fmadd(int,CiRegister,CiRegister,CiRegister,CiRegister)
	fmsub(int,CiRegister,CiRegister,CiRegister,CiRegister)
	fcmp(int,CiRegister,CiRegister)
	fcmpInstruction(CiRegister,CiRegister,InstructionType)
	fccmp(int,CiRegister,CiRegister,int,ConditionFlag)
	fcmpZero(int,CiRegister)
	fcmpZeroInstruction(CiRegister,InstructionType)
	fcsel(int,CiRegister,CiRegister,CiRegister,ConditionFlag)
System Register Access	mrs(CiRegister,SystemRegister)
	msr(SystemRegister,CiRegister)
	msr(PStateField,int)
Halting Debug	hlt()
End of Table	

Bibliography

[Apa05] Apache. Apache harmony. https://harmony.apache.org/, 2005.

[ARM11] ARM. Arm discloses technical details of the next version of the arm architecture. http://www.arm.com/about/newsroom/arm-discloses-technical-details-of-the-next-version-of-the-arm-ar php, 2011.

[ARM12] ARM. Architecture reference manual. armv7-a and armv7-r edition. *ARM DDI C*, 406, 2012.

[ARM15a] ARM. Arm architecture reference manual - armv8, for armv8-a architecture profile, 2015.

[ARM15b] ARM. Arm cortex-a series programmer's guide for armv8-a, 2015.

[Blu96] William Blundon. How can java rule the world? http://www.javaworld.com/article/2077139/how-can-java-rule-the-world-.html, 1996.

[DBR02] Scott W Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture, May 28 2002. US Patent 6,397,242.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[Goo11] John Goodacre. Technology preview: The armv8 architecture. *White Paper, Nov*, 2011.

[Gou11] Artiom Gourevitch. Just in time compiler (jit) in hotspot. `http://java.dzone.com/articles/just-time-compiler-jit-hotspot`, 2011.

[Gri11] Richard Grisenthwaite. Armv8 technology preview, 2011.

[Had15] Hadoop. Poweredby. `http://wiki.apache.org/hadoop/PoweredBy`, 2015.

[Har14] Tobias Hartmann. Code cache optimizations for dynamically compiled languages. 2014.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7, 2008.

[Lin14] Linaro. Linaro gcc for armv8. `https://www.linaro.org/projects/armv8/`, 2014.

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[OG01] C Enrique Ortiz and Eric Giguère. *Mobile information device profile for Java 2 MicroEdition: professional developer's guide*, volume 15. John Wiley & Sons, 2001.

[Ope15] OpenJDK. The hotspot group. `http://openjdk.java.net/groups/hotspot/`, 2015.

[PTHH14] Tobias Pape, Arian Treffer, Robert Hirschfeld, and Michael Haupt. *Extending a Java Virtual Machine to Dynamic Object-oriented Languages*, volume 82. Universitätsverlag Potsdam, 2014.

[Šev08] Jaroslav Ševcık. The sun hotspot jvm does not conform with the java memory model. Technical report, 2008.

[Shi14] Anand Lal Shimpi. Arm partners ship 50 billion chips since 1991 - where did they go? `http://www.anandtech.com/show/7909/arm-partners-ship-50-billion-chips-since-1991-where-did-they-go`, 2014.

[Sim08] Doug Simon. Maxine vm. `https://kenai.com/projects/maxine`, 2008.

[Sim11a] Doug Simon. The maxine project: Assemblers. `https://bigip-wikis-cms-adc.oracle.com/display/MaxineVM/Assembler`, 2011.

[Sim11b] Doug Simon. Maxine vm project home. `https://bigip-wikis-cms-adc.oracle.com/display/MaxineVM/Home`, 2011.

[SSB12] Robert F Stärk, Joachim Schmid, and Egon Börger. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media, 2012.

[TIO15] TIOBE. Tiobe index for september 2015. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, 2015.

[Ven96] Bill Venners. *Inside the Java virtual machine*. McGraw-Hill, Inc., 1996.

[WF98] Dan S Wallach and Edward W Felten. Understanding java stack inspection. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 52–63. IEEE, 1998.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):30, 2013.

[WW12] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14. ACM, 2012.

[ZR13] Yuhao Zhu and Vijay Janapa Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 13–24. IEEE, 2013.