Virtual Machines: Features and Futures

by
Brian Robert Hirshman

A Thesis
Submitted in partial fulfillment of
the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts
May 8, 2006

Contents

1 Introduction **1**
- 1.1 Current Technology . . . 1
- 1.2 Future Unification . . . 1
- 1.3 How To Read This Thesis . . . 2

2 Virtual Machine Support for Languages **3**
- 2.1 Lisp & Scheme . . . 3
- 2.2 Pascal . . . 5
- 2.3 Smalltalk . . . 6
- 2.4 Self . . . 7
- 2.5 Java . . . 8
- 2.6 Perl & Parrot . . . 9
- 2.7 Python . . . 11
- 2.8 CLR and .NET . . . 12
- 2.9 Other Virtual Machines . . . 13

3 Features of Virtual Machines **16**
- 3.1 Platform Independence . . . 16
- 3.2 Paravirtualization . . . 18
- 3.3 Migration . . . 20
- 3.4 Dynamic Optimization . . . 22
- 3.5 Security . . . 23
- 3.6 Development . . . 25

4 The ++VM **27**
- 4.1 Overview . . . 27
- 4.2 Tags . . . 27
- 4.3 Memory Management . . . 30

4.4 Objects 31
4.5 Opcodes 32
4.6 Attributes & Annotations 33

5 Object and Object Attribute Support 34
5.1 Object Layout 35
5.2 Object Behavior 38
5.3 Object Attributes 40
5.4 Features of Objects 41

6 Intermediate Value Support 45
6.1 Registers 45
6.2 Register Windows 51
6.3 Features of Registers 53
6.4 Memory 55
6.5 Intermediate Value Movement 57
6.6 Features of Memory 59

7 Tagged Memory 60
7.1 Tag Implementations 60
7.2 Tag Usage 61
7.3 Tag Features 63

8 Opcodes 66
8.1 Opcode Format 67
8.2 Opcode Execution 68
8.3 Bytecode Instructions 69
8.4 Control Operations 70
8.5 Features of Opcodes 73

9 Methods 77
9.1 Methods in the ++VM 77
9.2 Kinds of Methods 77
9.3 Method Behavior 79
9.4 Method Attributes 80
9.5 Returning From Methods 81
9.6 Return Behavior 82
9.7 Wound Calls 82

9.8 Features of Methods 84

10 Code Support **86**
10.1 Opcodes with Attributes 86
10.2 Features of Attributes 88
10.3 Annotation Opcodes 89
10.4 Adopted Annotations 91
10.5 Features of Annotation Opcodes 91

11 Code Groups **94**
11.1 Code Blocks 94
11.2 Accessing Libraries 97
11.3 Using Code Groups 99
11.4 Features of Code Groups 100

12 Control Flow **103**
12.1 Logical Control in the ++VM 104
12.2 Features of Logical Control 105
12.3 Loops in the ++VM 106
12.4 Features of Loops 108
12.5 Exceptions in the ++VM 110
12.6 Handling Exceptions 111
12.7 Features of the Exception Mechanism 113

13 Conclusion **114**
13.1 Features of the ++VM 114
13.2 Future Work 118

A Tags **120**
A.1 Byte 121
A.2 Short 122
A.3 Word 123
A.4 Long 124
A.5 Ultra 125
A.6 Float 126
A.7 Double 127
A.8 Reserved 128
A.9 Standard Object 129

A.10 Object with Code 130
A.11 List Element 131
A.12 Class Object 132
A.13 Future Object 133
A.14 Future Object with Code 134
A.15 Pointer 135
A.16 Reserved 136

B Opcodes **137**
B.1 ILLEGAL INSTRUCTION [ILL] 139
B.2 STACK OPERATIONS [STK] 140
B.3 ADDITION [ADD] 142
B.4 SUBTRACTION [SUB] 144
B.5 MULTIPLICATION [MUL] 146
B.6 DIVISION [DIV] 148
B.7 MODULUS [MOD] 150
B.8 NUMERIC INNER PRODUCT [NPR] 152
B.9 LOGICAL SHIFT RIGHT [LSR] 153
B.10 ARITHMETIC SHIFT RIGHT [LSR] 155
B.11 LOGICAL SHIFT LEFT [LSL] 157
B.12 BIT OPERATIONS [BOP] 159
B.13 SET UNION [UNN] 161
B.14 SET INTERSECTION [NTR] 163
B.15 SET EXCLUSIVE OR [XOR] 165
B.16 SET INNER PRODUCT [SPR] 167
B.17 CONVERT [CON] 168
B.18 CAST [CST] 169
B.19 ONE OPERAND INSTRUCTIONS [OOP] 171
B.20 MOVE BETWEEN REGISTERS [MRG] 173
B.21 LOAD FROM OBJECT OFFSET [LOB] 175
B.22 STORE TO OBJECT OFFSET [SOB] 177
B.23 LOAD FROM POINTER OFFSET [LPT] 179
B.24 STORE TO POINTER OFFSET [SPT] 181
B.25 PREPARE NEW OBJECT [NEW] 183
B.26 OBJECT ATTRIBUTE [ATR] 185
B.27 CREATE A METHOD [MTH] 188
B.28 CREATE A CLASS [CLS] 190
B.29 BLOCK START [SBK] 192

B.30 BLOCK END [EBK] 195
B.31 THROW AN EXCEPTION [TRW] 197
B.32 CATCH AN EXCEPTION [CAT] 199
B.33 COMPARE DATA VALUES [CPD] 201
B.34 COMPARE REFERENCE VALUES [CPR] 203
B.35 COMPARE BOUNDS DATA [CBD] 204
B.36 COMPARE BOUNDS REFERENCE [CBR] 205
B.37 CONDITIONAL BRANCH [BNZ] 206
B.38 CONDITIONAL BRANCH [BXC] 208
B.39 CALL METHOD [CAL] 210
B.40 RETURN FROM METHOD CALL [RTN] 212
B.41 ENTER A MONITOR [MEN] 214
B.42 EXIT A MONITOR [MEX] 216
B.43 YIELD PROCESSOR CONTROL [YLD] 217
B.44 SYNCHRONIZATION POINT [SYN] 218
B.45 ANNOTATION [ANO] 219
B.46 LOOP [LUP] 221

Chapter 1

Introduction

As yet, there is no formal science of designing virtual machines. Instead, virtual machine design has been ad hoc, a reaction to the current mismatch between hardware and software. This thesis develops a specification for a thin, general-purpose virtual machine that is sensitive to the needs of programmers as well as hardware designers. Some readers may consider this thesis to be a blueprint for a novel target virtual machine. Others may see this specification as a model for future hardware. By studying the ++VM virtual machine, introduced in later chapters, this thesis examines the strengths and establishes goals of virtual machine design.

1.1 Current Technology

For forty years, virtual machines have provided features that extend the capabilities of simple hardware. Recently, though, virtual machines have begun to provide ever more complex features, addressing problems such as security and platform independence. These features, and many others, are addressed in Chapter 3. High-level languages have often targeted virtual machines to take advantage of these features. Chapter 2 addresses several existing virtual machines, including Lisp, Pascal, Java, Perl, and Python.

1.2 Future Unification

The remainder of this work specifies a new high-level language virtual machine, the ++VM. The ++VM is designed to be broad and general purpose virtual machine

and is not designed to be targeted by one particular language or one particular architecture. The ++VM is designed to be flexible enough to support a diverse collection of languages. In doing so, it requires a number of features not found in high-level language virtual machines today. Chapter 4 provides a brief overview of this virtual machine.

The ++VM virtual machine creates a modern memory environment that provides effective support for current high-level languages. Chapter 5 discusses object-oriented support in the ++VM, while Chapter 6 addresses registers and other intermediate values. Chapter 7 addresses the virtual machine's tagged memory implementation, a system designed to support both objects and intermediate values.

The ++VM uses a minimal instruction set. More information about the design of the instruction can be found in Chapter 8, including the opcode layout and the kinds of instructions that can be performed. Additional detail on methods, and method invocation in particular, can be found in Chapter 9.

The ++VM also provides support for a number of advanced control structures. Generally, these are brought about by the use of code attributes, as described in Chapter 10, or via code groups, as described in Chapter 11. When combined, three features allow for a more informed manner of addressing logical control flow. Chapter 12 addresses these control flow issues like logic operations, loops, and exception handling. By giving more compile-time information to the virtual machine, all of these components allow the virtual machine to perform more efficiently.

This work concludes with a brief discussion of how hardware and software can take advantage of the ++VM or other virtual machines.

1.3 How To Read This Thesis

There are many ways readers may find this work useful. Readers may wish to use the table of contents to read about a particular aspect of the ++VM without examining other features in detail. Other readers may wish to skip straight to the appendices to find a bitwise description of the function that each ++VM opcode performs. Still other readers may wish to skim the end of every chapter to understand the arguments for specific design decisions. This thesis is designed to support all of these approaches. Readers who choose to read only selections from this thesis may find it useful to skim the overview chapter, Chapter 4, before reading specific chapters of interest, as Chapter 4 presents a broad yet succinct overview of the ++VM.

Chapter 2

Virtual Machine Support for Languages

In order to understand the state of current high-level language virtual machines, it is necessary to conduct a comparative examination of different high-level language virtual machines. Though the implementation of current high-level language machines is vastly different than machines forty years ago, they are conceptually very similar. Important languages in the history of virtual machines include Lisp, Pascal, Self, Smalltalk, Java, Perl, Python, and .NET.

Though this survey of past virtual machines focuses primarily on high-level language virtual machines, virtual machines have been used to attack a variety of other problems. The concluding section of this chapter outlines other potential uses for virtual machines.

2.1 Lisp & Scheme

2.1.1 Language

The Lisp language was perhaps the oldest language to run on a virtual machine. The first Lisp programs were hand-compiled by John McCarthy at MIT in 1960, and were meant to run specifically on the IBM 704 hardware. Since the Lisp language had a powerful evaluation operation, it was more flexible than previously developed programming languages. Since McCarthy's initial Lisp design, the language has evolved and split into a number of variants differentiated by syntax and features. Current

variants of Lisp include the Lisp precursor FLPL, developed in the late 1950s, the later Franz Lisp of the early 1980s, and perhaps the most well-known variant, Scheme, developed in 1975 by Sussman and Steele. Though Lisp was originally created with a target architecture in mind, the language's elegant syntax long outlived the hardware for which it was designed. When that hardware, the IBM 704, disappeared, Lisp programmers wrote virtual machines that emulated the 704's architecture style to continue using the Lisp language. Over time, Lisp virtual machines have been implemented in a number of languages: early FLPL Lisp virtual machines were implemented in Fortran, while later virtual machines for other variants have been implemented in languages such as C [Joh78, Fat81, SG96].

2.1.2 Virtual Machine

The Lisp virtual machine was a register-based machine, with every "cell" in memory containing information in a car register and a related pointer in a cdr register. The instruction set for the Lisp virtual machine was a direct outgrowth of the 704's instruction set. As the 704 instruction set was relatively simple, the Lisp virtual machine itself is relatively straightforward. Some variants of the Lisp language can be implemented using a dozen or so opcodes, though implementations with side effects require a handful of additional instructions [Fat81]. The majority of the Lisp opcodes are devoted to data movement and method calling. Since the Lisp system was originally designed to mimic hardware behavior, the Lisp opcodes were only capable of providing only low-level support for high-level language functions.

The Lisp virtual machine introduced a number of features that have been used in many other subsequent virtual machines. Lisp allowed code to be interpreted at runtime as opposed to being statically compiled, as its "eval" operation allowed the virtual machine to evaluate a dynamically created expression. This allowed the Lisp machine to have an elegant execution mechanism, and allowed the virtual machine to easily support self-modifying code. Additionally, the Lisp virtual machine also introduced a primitive form of just-in-time compilation. Later Lisp variants allowed, and indeed encouraged, their implementers to compile Lisp code into machine-specific languages as the code was executed. Additionally, the Lisp virtual machine was also the first to introduce a garbage collection mechanism. Garbage collectors, found in the vast majority of later virtual machines, free the programmer from having to worry about dynamic memory allocation [Joh78, Ayc03].

2.2 Pascal

2.2.1 Language

The Pascal language was developed in 1970, by Niklaus Wirth. It was based on the Algol programming language, and was initially designed for the CDC 6000 computer family. However, Pascal quickly outgrew the single system for which it was designed, and eventually was made into a platform-independent language with the introduction of P-code. Though P-code was originally designed to simplify the compilation of Pascal, it was not long before there were three platform-independent implementation of Pascal P-code. Standard P-code or P-4 code, the first of these systems, is a straightforward stack-based computation system. UCSD P-code was designed for the execution of Pascal code on a small computer and was a very space- and memory-conscious implementation. LASL P-code, the last of the systems, was designed for large applications that would be time sensitive. These three systems helped Pascal to gain widespread acceptance by the end of the decade [Wir93, Nel79].

2.2.2 Virtual Machine

The Pascal P-code virtual machine is a stack-based machine and runs using an instruction set of about sixty instructions. Stack-based machines like the Pascal virtual machine execute instructions by pulling elements off of an internal storage area, performing an operation, then pushing values back; many machines, the Pascal machine among them, do not make registers available for user values. While the majority of the Pascal instructions deal with memory movement or mathematical operations, there are several instructions which perform relatively abstract operations such as testing for the end of a file. Pascal has a number of unusual opcodes that were not used in previous virtual machines and have rarely been used in subsequent ones. These include operations that deal with sets, bit fields, and files, in addition to a number of Pascal-specific addressing modes [PD82].

The virtual machine that ran the Pascal code introduced a number of new features to high-level language computing. First, the Pascal virtual machine was the first virtual machine to be designed for an imaginary architecture and not for real hardware. This made it possible to run Pascal on any system that implemented a virtual machine interpreter, regardless of the underlying chip, and this made the language easily portable. Additionally, UCSD P-code was the first virtual machine language to have code elements of a standard size, bytes, which led to more compact code storage in memory. This RISC-like design decision would be adopted by almost all virtual

machine instruction sets in the future as it made instructions easier to understand. Furthermore, the Pascal machine was the first machine to have primitive mathematical operations as part of its instruction set. It was not until the development of the Pascal virtual machine that primitive math operations were included as a virtual machine instruction as opposed to being shipped off to the host system via a method call. This may have been related to the fact that the P-code machine was the first virtual machine designed to be platform-independent, as it was not designed to use native instructions. [Nel79, PD82].

2.3 Smalltalk

2.3.1 Language

Smalltalk was designed at Xerox PARC by a team of researchers including Dan Ingalls in the 1970s. In contrast to Pascal, which was an imperative programming language, Smalltalk was object-oriented. Smalltalk is a big system. Smalltalk was meant to be a complete and independent programming environment, and for this reason needed to duplicate much of the operating system's functionality in order to achieve complete platform independence. This meant that, in many ways, Smalltalk acted much more like a hosted virtual machine than a high-level language one: much of the underlying system was simply ignored by the Smalltalk environment and re-written using the virtual machine. Perhaps the most well known current current Smalltalk implementation is Squeak, a small Smalltalk implementation developed in...Smalltalk [GR83b].

2.3.2 Virtual Machine

The Smalltalk virtual machine was a stack-based virtual machine, similar to the machine used to run Pascal. The Smalltalk virtual machine used a bytecode, allowing it to have 256 opcodes to control code execution. The vast majority of the Smalltalk instructions manipulate the internal stack by pushing, popping, or manipulating stack values. Additionally, many operations are related. A sizable percentage of Smalltalk instructions are optimized instances of more general instructions; in fact, the language defined less than fifty unique instructions. Smalltalk, like Pascal, provided support for a number of primitive math operations; unlike Pascal, it allowed the behavior of some virtual machine instructions to be changed with compiler modifications [GR83b, GR83a].

The creation of the Smalltalk virtual machine marked a large shift in the role of

the virtual machine. First, Smalltalk was the first object-oriented virtual machine. This meant that every data element in the virtual machine was stored and addressed using a common object format. In addition, the Smalltalk compiler could modify the behavior of some of the virtual machine bytecodes. Because of this, Smalltalk allowed programmers to custom-tailor their environment with specific optimizations. Additionally, the Smalltalk environment was meant to be dynamically reconfigurable. Every Smalltalk method would be called via a message-passing scheme, and methods could be dynamically changed at runtime. Furthermore, the Smalltalk virtual machine took on roles that were traditionally those of the operating system. The Smalltalk machine sought to form a completely independent programming environment, and to do so even sought to create a platform-independent graphics kernel, BitBlt. The message-passing scheme necessitated the development of much more efficient just-in-time compilers. These rapid JITs were needed in order to achieve adequate system performance, as interpreting Smalltalk opcodes was inordinately slow if not downright impossible [WBC99, GR83b, Ayc03].

2.4 Self

2.4.1 Language

The Self language, developed in the late 1980s, was partially inspired by Smalltalk. Like Smalltalk, Self was object oriented; unlike Smalltalk, Self was developed without classes, meaning that every object could define its own behaviors. This was a rather radical design decision, and posed a number of difficulties for the language implementers who had do deal with the fact that systems might not have enough memory to run inefficient implementations of the Self virtual machine. Self may not be as well known as other high-level languages that run on virtual machines; the most notable implementation of the Self system may the C++ Chambers implementation in the early 1990s [ABC$^+$, CUL89].

2.4.2 Virtual Machine

The self language runs on a stack-based virtual machine machine, similar to the type of machine used to run Smalltalk. In contrast to Smalltalk, the Self virtual machine operated using only eight unique instructions! These opcodes were designed to be as flexible and as elegant as possible, even at the cost of being potentially less efficient than opcodes in other languages because of their lack of specificity. Self used a byte to

hold every instruction, the low five bits of each opcode were used to store field indices to facilitate memory loads. Of the eight opcode families in Self, four were used to send messages between one object and another. By sending messages back and forth, objects could invoke methods (such as addition) in order to compute intermediate and final values [CUL89].

The Self virtual machine marked an interesting progression in the evolution of the virtual machine. First, the Self virtual machine introduced the idea of per-object annotations. Though these annotations provided information to the programmer - and not to the runtime environment - they provided a way for programmers to provide optional information about specific objects. Additionally, the Self machine was the first major virtual machine to use multiple bits to tag memory. The Chambers implementation of the Self virtual machine used four bits to tag memory, using tag bits to distinguish between different types of data and pointer values. Lastly, the Self virtual machine spurred another round of improvements in just-in-time garbage collectors. This improvement helped to mitigate some of the slowdown related to the design decisions of the language, though it took several generations worth of compiler and just-in-time compiler design in order to create an effective system [ABC$^+$, CUL89, Ayc03]

2.5 Java

2.5.1 Language

The Java language was designed to meet a profoundly different need than earlier languages: Java was designed to be a robust and secure system that would be portable. Java was created at Sun in the early 1990s, the brainchild of a number of engineers including James Gosling. Java was meant to run a secure environment that could be easily ported to many different systems. One of the major causes of Java's success was the desire to run potentially untrusted code on the Internet; Java's security system, one of the virtual machine's major innovations, allowed any user to run a untrusted program safely. Java has gone through a half-dozen major revisions since its inception, as more features and additional functionality were steadily added to the language; however, these changes have not significantly changed the underlying opcodes or the virtual machine. The most interesting Java virtual machine from a researcher's point of view is IBM's Jikes virtual machine, a freely-available Java virtual machine which has pioneered a number of changes to Java virtual machine design [GM96, Mac05].

2.5.2 Virtual Machine

Like the Pascal virtual machine, the Java virtual machine is a stack-based virtual machine. The Java virtual machine stores each of its opcodes in a single byte, and defines two hundred different opcodes for use in the virtual machine. The vast majority of the Java opcodes are for data movement or mathematical operations. The primary reason for the preponderance of math opcodes is due to the fact that every Java opcode specifies the type of its operands, a property that allows the virtual machine to easily verify the type safety of the code which it is about to execute. Java also included a native method interface, a process by which Java programs could call methods written in other languages and run code outside of the confines of the virtual machine [GM96].

The Java language introduced a number of new features to the virtual machine community. As one of the first major high-level languages to stress security, Java needed to have a virtual machine which was able to enforce the boundaries that the language sought to impose. This was accomplished using a verifier, a system that explicitly ensured that, for instance, a method would not reference a variable to which it should not. To take advantage of this security policy, the Java opcode set was designed for transport over the network. The compactness of Java's class structure and opcode set, as well as the importance of networking classes to the standard virtual machine library, helped to make Java a leading environment for web development. Yet another innovation was the fact that Java included a system for invoking non-Java methods. This was done using a native method interface which allowed Java methods to call non-Java methods and vice-versa. Furthermore, Java was also among the first languages designed to be dynamically modifiable. One of its major innovations of Java is that the runtime could be dynamically modified as new classes to be loaded or defined while code was being execute. Lastly, the Java virtual machine was the first virtual machine that was widely used by non-programmers due to its appearance on the Internet. The Java virtual machine was partially responsible for a resurgence of interest in virtual machines, and, for better or worse, it is probably the most widely known virtual machine in use today [GM96, LY99, BCF+99].

2.6 Perl & Parrot

2.6.1 Language

Unlike languages such as Smalltalk or Java, Perl was initially designed to be a small scripting language, though over time it has expanded to become a much more complete

language. Perl was developed by Larry Wall in 1987, and one of its fundamental design goals was to make the "easy things easy and the hard things possible." In order to do so, the Perl environment allowed users to write code in an English-like manner and to be as flexible as possible. The Perl6 implementation of the language, also known as Parrot, includes a virtual machine that is a near-complete rewrite of the Perl virtual machine, and involved a re-definition Perl data types and opcodes. As part of the virtual machine rewrite, the Parrot development team attempted to combine Perl opcodes with those of other scripting languages in order to make the Parrot system capable of running a wide variety of languages [Ash05, Org05b, Sug02].

2.6.2 Virtual Machine

The Parrot Virtual machine is register based, like the Lisp virtual machine. Unlike Lisp, the Parrot virtual machine is capable of defining an arbitrary number of registers - and can actually define more registers than the host machine actually has - in order to store intermediate values. Parrot also uses seven backup stacks to hold values that are not held in registers. Additionally, the Parrot implementation of Perl has a relatively large instruction set. Parrot uses a 32-bit opcode format, which allows for a much larger instruction set than any previously seen virtual machine language. The vast majority of these instructions were used to type specific operations, as the Perl virtual machine typed its instructions in order to improve overall performance. However, this large instruction set allows Perl to have instructions not found in other virtual machines, such as opcodes to control the behavior of the garbage collector and for string manipulation [Org05b].

There were a large number of virtual machine features that also differentiate Perl from other languages. Parrot's execution model for Perl6 is a model that contains seven stacks and a large number of registers; unlike earlier systems that were either purely stack based or register based, Parrot attempts to take advantage of both approaches. Additionally, Parrot tries to treat exceptions and loops in the same way, by using similar instructions to take advantage of the similarity of execution style. Furthermore, though it is not necessary to assign types to variables when writing Perl code, the Perl compiler and the Parrot virtual machine assign types to these variables and optimize them variables in type-specific ways. Lastly, the Parrot virtual machine has two intermediate languages, not one. Though a human user would be exposed to a complex instruction set as the base programming level in Perl, the opcodes executed by the virtual machine were actually those of a much simpler form. The Parrot Assembly Language, as the programmer-level code was called, allowed for finely-tuned hand coding in Perl and could lead to significant performance increases in places where

humans could provide more run-time information than the compiler could glean from standard Perl code. The combination of all of these virtual machine features helped to make Perl an important language for short scripting applications. [Ash05, Org05b].

2.7 Python

2.7.1 Language

Python - the name refers to Monty Python's Flying Circus, not to the snake - was created in 1991 by Guido van Rossum as a scripting language. Like Perl, Python was meant to be a fast and easy language in which to write code, and its syntax is much less restrictive than that of other languages. The original Python virtual machine was written in C, though later virtual machines have been written in other languages for speed or security. Free versions of Python are available from the Python Software Foundation [Mar03, Org03].

2.7.2 Virtual Machine

The Python virtual machine is a stack-based machine, like Java and Pascal. Unlike Java, the Python stack only holds references; should a program wish to perform a math operation on a primitive value, a reference to that value will be pushed on the Python stack, not the primitive itself. The Python virtual machine has about the same number of instructions as Java and many fewer than Parrot. Much of the decrease in the number of Python instructions is due to the fact that Python does not require the same level of type information as is needed by the Java and Parrot virtual machines. The instruction set has opcodes to explicitly build new classes and methods [Org03, Hug97].

The Python virtual machine introduced a number of interesting features. First, it provided opcode support for complex structures such as loops and iterators, making them an integral part of the virtual machine. This provided additional high-level information to the virtual machine and resulted in a significant increase in loop performance. It also introduced the concept of in-place operations, operations that were performed directly on the stack rather than popped off, computed, and pushed back on. This improved performance. Lastly, Python also allowed "docstrings" into the code, comments that could be attached to a Python objects at runtime. Like the annotation system provided in Self, these docstrings provided a way for programmers to include debugging information in runtime objects [Org03, YvR01].

2.8 CLR and .NET

2.8.1 Language

While .NET began as Microsoft's response to Sun's Java, it rapidly evolved to be a more complicated form of virtual machine environment. The .NET environment, originally created in 1998, used a common intermediate language to allow the virtual machine to run multiple languages. In contrast to the Parrot project for Perl, not all the languages that were brought under the .NET umbrella were traditionally run on virtual machines. Some of these languages, such as Visual Basic and C#, had to be slightly modified to work in the new system. In attempting to bring together so many languages, Microsoft abandoned some of the functionality of each language but in so doing created a system which had minimal conflicts with existing code. The resulting product, which ran on a virtual machine known as the Common Language Runtime or CLR, extended the functionality of virtual machines into new and potentially interesting domains [Cor05b].

2.8.2 Virtual Machine

The .NET virtual machine, called the CLR, used a stack architecture to hold intermediate data values. The intermediate language that is executed on the CLR, called the Common Intermediate Language or CIL, has about three hundred instructions. Even though the CIL used a single byte to hold each instruction, it manages to have more than 256 instructions because it uses one instruction to signal an extended instruction, an instruction represented using multiple instruction bytes. The CIL introduced a number of interesting operations, including explicit tail calls for recursive methods and methods to explicitly box and unbox primitives [NR302a].

The Common Language Runtime was a marked departure from past virtual machines for a variety of reasons. For one, the CLR was the first virtual machine that was designed with the direct intention of supporting multiple languages. This design decision added a whole new level of complication to the virtual machine, though eventually the CLR supported almost all the language features of the half-dozen languages that it was designed to support. The CLR allowed programmers to write code in one .NET-supported language, then read or modify these objects using code written in another. Another new feature of the .NET virtual machine was the ability to directly access to memory via pointers. While past virtual machines had made memory a completely abstract entity, .NET permitted directly to the underlying data structures if the code had sufficient privileges. Lastly, the behavior of the .NET verifier and

garbage collector could be firmly controlled by the programmer. These capabilities broke some of the traditional bounds of virtual machine design, giving more freedom to the programmer at the expense of potintially unsafe code [Cor05b, NR302a].

2.9 Other Virtual Machines

Though the majority of this work focuses on virtual machine support for high-level languages, virtual machines have traditionally been used in a number of other realms. This section highlights a number of ways in which virtual machines have been used to support computation, including operating systems, hosted virtual machines, co-designed virtual machines, and multi-programmed systems.

2.9.1 Operating Systems

The most common type of virtual machine is the operating system. Though normal computer users might not think of the operating system as a form of virtual machine, it does provide many of the basic features of described above: it provides a common set of operating system calls to guest programs, enforces a security policy, and handles threading and synchronization. The operating system is so fundamental to the modern concept of the computer that it is not often thought of as a virtual system. From the perspective of a process running on the machine, however, much of the hardware has been abstracted by the procedure calls of the operating system. A process does not need to have a detailed understanding of the type of disk being accessed; it only needs to know that an operating system provides one. Similarly, an operating system enforces a security policy, by restricting file and memory access to certain programs and users. If a user program attempt to access data improperly, the operating system will cause that program to fail. Lastly, the operating system provides kernel threads, allowing for multiple streams of execution to occur on the processor. Threading libraries, such as pthreads, move thread support from the the program layer to the layer below it [SN05].

Implementations of operating systems are both abundant and familiar, and are the subject of numerous investigations. However, it may surprise the reader to think that, without the help of the virtual machine currently running, this document could not have been read or written!

2.9.2 Hosted Virtual Machines

A hosted virtual machine is designed to allow a system that uses one instruction set to run on a system that uses another. When operating systems are designed, they are optimized for a particular hardware instruction set architecture. Should a user wish to run that operating system on a different type of hardware, a virtual machine will be necessary to convert between the two architectures. The virtual machine will present the guest operating system, the one that the user wishes to run, with an architecture that feels like the one on which it is supposed to run; however, the host system for the virtual machine will only execute its own instructions. The virtual machine can get around this language barrier by emulating or translating from the guest instruction set to the host's instruction set, should the guest system attempt to execute an instruction that cannot be immediately executed on the host system [SN05].

For instance, older versions of the Macintosh operating system will now be run exclusively on hosted virtual machines. In June of 2005, Apple decided to abandon its PowerPC chip architecture and use Intel hardware as the target for its operating system. However, the Macintosh operating system, for most of the last decade, has been designed for the PowerPC processor. In order to ensure backwards compatibility between programs designed for its old and new systems, Apple hired Transitive Corporation to build a PowerPC emulator called Rosetta, a hosted virtual machine that does binary translation to convert between PowerPC and Intel instructions [SAP05, Cor05a].

2.9.3 Co-designed Virtual Machines

Co-designed virtual machines are designed to run a new, innovative instruction set architectures beneath a more compex or older instruciton set. Co-designed virtual machines exist on the hardware chip itself and modify the instructions that the hardware will execute. One use of co-designed virtual machines is in breaking down complex instructions, such as those of the Intel x86 instruction set, into smaller units that can be rapidly executed on the processor. The Intel instruction set architecture is a relic of the 1980s, and many of its instructions are not optimized for pipelining or simultaneous execution on modern processors. When breaking down complex instructions, co-designed virtual machines can take advantage of the virtual machine optimizations to further increase performance. These optimizations can boost speed and reduce power consumption, allowing for faster and more efficient processing of instructions [SN05, Kla00].

One of the most well known co-designed virtual machines is the co-designed vir-

tual machine at the core of the Transmeta Crusoe processor. The Crusoe processor converts Intel instructions to smaller instruction units, called molecules, allowing four different CPU operations to proceed simultaneously. This facilitates out-of-order instruction execution, caching, and branch prediction, all of which will increase speed. The code morphing software at the heart of the processor allows for the processor to execute instructions more quickly and substantially save power. Many of the innovations in co-designed virtual machines, which form the heart of the Crusoe processor, have been adopted by Intel and other chip makers [Kla00, Cor05c].

2.9.4 Multiprogrammed Systems

Multiprogrammed systems are designed to increase the performance of previously compiled programs, even programs designed to run on the same instruction set architecture as the host machine. A multiprogrammed system will take code that compiled to executable form and re-optimize it to make it run faster. Though dynamically recompiling code may seem like a waste of time and space, it can result in a performance increase. When a program is compiled statically, the compiler will not be able to determine the characteristics of the program's run-time behavior. A multiprogrammed system, however, will be able to gather run-time information by taking advantage of the profiling capabilities of a virtual machine. The multiprogrammed system can then take advantage of optimization techniques of Section 3.4, such as block formation and code reordering, to incorporating this run-time information into the code. A multiprogrammed system should be able to achieve better performance than running the statically-compiled program in a standard environment. If implemented efficiently, these modifications to the original code will more than make up for the overhead of supporting the virtual machine [SN05].

Examples of multiprogrammed systems include various optimization engines such as the Dynamo project at HP labs and the DynamRIO project at MIT. These systems attempt to optimize code performance as code is being executed, and in some cases managed to achieve a 25% increase in benchmark performance. Because of the possible performance improvement, some operating system designers are beginning to thinking about including multiprogrammed system virtual machine techniques in the creation of future operating systems [SN05].

Chapter 3

Features of Virtual Machines

Over the last forty years, hardware and software designers have come up with a number of ways in which virtual machines can help manage this complexity. This chapter highlights a number of virtual machine features, and highlights a number of past and active research projects to take advantage of these features. Though this survey of virtual machine features focuses primarily on high-level language virtual machines, may of these features are useful in other types of virtual machines as well. While the majority of the examples chosen in this chapter focus on applications to high-level language virtual machines, many of these advantages apply to these additional domains.

This chapter highlights past research in ways that virtual machines can help to achieve platform independence, to support paravirtualization, to facilitate environment migration, to optimize code dynamically, to provide better security apparatus, and to shorten hardware and software development cycles.

3.1 Platform Independence

Virtual machines provide platform independence, the ability to free a program from a particular host machine, by isolating the running code from the vagaries of host software and hardware.

3.1.1 Software Isolation

Virtual machines often must provide support for an interface that would normally be the responsibility host software such as the operating system. In order for a virtual machine to be fully independent, its features must be available in all implementations on all hosts. In practice, this means that virtual machines will commonly duplicate much of the functionality of the host's operating system. For instance, many high-level language virtual machines define a policy for handling threads and assigning priorities to processes, meaning that every virtual machine for that language includes a thread library as an integral component. Other virtual machines have special ways of opening and reading files in order to ensure that file reading is consistent regardless of the host type. By replicating operating system behavior in this way, virtual machines can ensure uniform behavior across multiple types of operating systems [SN05].

In some machines, the level of specification is more rigorous than in others. For instance, in the Java language, general thread behavior is specified but some scheduling details are left to the implementor. On the other hand, Smalltalk is completely specified to the point that virtual machine designers are required to meticulously implement a graphics kernel known as BitBlt. Though many virtual systems do not go quite as far in other areas, most virtual machines have at least a standard way of interfacing with the underlying system that allows for a degree of standardization in operating system calls. Though using a call to any operating system is potentially non-portable, virtual machines are able to achieve a large degree of uniformity by restricting their guest programs to making relatively simple, commonly-supported calls [LY99, GR83b].

3.1.2 Hardware Isolation

Equally important to the notion of platform independence is the idea that the virtual machine insulates the guest program from the hardware. Many virtual machines present guest programs with the instruction set of an abstract machine, but must rely on the properties of the real machine to run it. If the real machine does not directly support the behavior, the virtual machine is required to emulate the instruction and break it down into smaller sections that can be run on the host platform. In these cases, providing a consistent hardware environment allows the virtual machine to be platform independent and to run programs in a more consistent manner [SN05].

For instance, the Java programming language specifies a particular standard of floating-point arithmetic, ANSI 754, in order to ensure that a Java result will not be biased by the behavior of the underlying hardware. These types of specifications

occur in other virtual machines as well, and not just machines designed to run high-level languages. In other virtual machines it is necessary to emulate the true contents of certain status registers on the processor in order to achieve consistency. On virtual machines designed to host operating systems on Intel chips, the two-bit privilege level must be emulated to prevent a guest operating system from circumventing the virtual machine and accessing the hardware directly [LY99, Uea05].

3.2 Paravirtualization

Virtual machines can also be used to run multiple programs on one machine using a technique known as paravirtualization. Often, paravirtualization is used to run multiple copies of the same program on a single virtual machine: each process in a paravirtualized environment is led to believe that it has complete access to all computation resources, though in effect the underlying virtual machine allocate hardware resources as needed. Paravirtualization has major advantages: it can improve processor utilization, memory utilization, and device utilization.

3.2.1 Processor Utilization

Virtual machines such as paravirtualized environments are able to improve processor utilization. Before virtual machines came to prominence, running multiple copies of a program in a secure environment meant running multiple copies of an operating system; each of these operating systems had to run on a separate hardware system and needed its own processor. If each of these systems used the processor for only a small fraction of the time, then much of the machine computation time was wasted. Paravirtualization, on the other hand, allowed multiple copies of a process to run securely on the same system, allowing multiple copies of an operating system to coexist on a single hardware system. In this arrangement, a large number of systems could share a single processor, and, if scheduled appropriately, could increase resource utilization. When the process idles in a paravirtualized environment, then the virtual machine could replace it with a waiting process rather than waste computation cycles. This allows paravirtualized systems to increase the number of processes that can run on a single processor and increase processor use [WSG02].

Though processor utilization strategies have not been used in high-level virtual machines, this technique of eliminating idle loops has been used with great success in hosted virtual machines. For instance, the Denali project at the University of Washington is able to maximize processor use by running hundreds of lightweight

virtual processes on top of a hosted virtual machine without large degradations in performance [WSG02, SGG05].

3.2.2 Memory Utilization

Paravirtualized virtual machines can also improve memory use. Before paravirtualization, each copy of a process required nearly identical information about program state and operating system state. This meant that memory and disk space was largely identical across multiple machines. A paravirtualized environment, however, can hold multiple copies of the same program. If the system is savvy, it can combine identical data elements and save memory. For instance, a paravirtualized virtual machine can combine all identical memory pages into a single, global page reference. Additionally, each time a new machine is created, it can use a "copy on write" strategy in order to minimize the amount of memory it must copy to create a new environment within the virtual machine. All told, this technique can save a large amount of memory, vastly decreasing the cost of supporting an additional virtual machine: the savings from page sharing increases greatly in proportion to the number of supported processes, and can rise to nearly seventy percent in some systems.

Though these memory reuse techniques have not been incorporated into high-level language virtual machines, these techniques are heavily used especially in Xen. The Xen system, developed at Cambridge, is designed to support several copies of relatively large programs. It has also been used effectively in VMware's ESX server with a large degree of success. Both systems have shown that, when using efficient memory sharing techniques, the marginal memory cost of adding an additional process can be very small [Wal02, BDF+03].

3.2.3 Resource Scheduling

Paravirtualized virtual machines can also be used for scheduling the use of resources. When processes on different physical devices have to use a shared resource, efficiently scheduling the use of that resource may be costly. If the processes both run inside a virtual machine, however, then the scheduling costs can be substantially decreased. Using virtual machines, it may be possible to schedule resources which previously were difficult to schedule efficiently. For instance, virtual machines can take advantage of disk reads and writes from multiple processes in order to schedule them in a way that minimizes disk head movement. Other virtual machines can open a single network connection for multiple virtual processes, rather than opening one connection for each separate process. One particularly interesting optimization is that of interprocess

communication: in the absence of virtual machines, separate computers would have to use the network to allow communication between two processes; using virtual processes, a virtual machine can use shared memory on the host machine to do so, saving a substantial amount of time. Though the scheduling algorithm itself may take a substantial amount of time to execute, the performance benefits gained through efficient scheduling often outweigh the cost of running a virtual machine [BDF+03, WCSG05].

Though high-level language virtual machines have not been used for resource scheduling, this technique has been used in system such as the Denali project at the University of Washington. Denali takes advantage of the fact that they receive I/O requests from many different processes and can schedule these requests in such a way that disk utilization is maximized. [BDF+03].

3.3 Migration

Virtual machines can allow one environment to run simultaneously on multiple machines, and even to move processes from one host machine to another in a process known as migration. Since a virtual machine presents an abstract environment to its guest programs, virtual machines can be transparently swapped between different hardware environments, a property that makes virtual machines very useful for distributed computing. Environment migration can be used for several purposes, including load balancing, machine swapping, and failure protection.

3.3.1 Load Balancing

As was addressed in the previous section, one of the major advantages of virtual machines is that they can combine a number of computationally inexpensive processes on the same physical hardware in order to achieve better performance. The converse is also true: virtual machines can be used to distribute the execution of computationally intensive processes across multiple machines in order to achieve better throughput. If some physical machines in a network are not being fully utilized, a virtual machine can realize this and reshuffle processes across different hardware systems. In this way, virtual machines can achieve more consistent levels of throughput in a network, rather than having some physical processors idle while others are overloaded. Since virtual machines allow computation to be done in a platform independent manner, the virtual machine will not have to care what type of hardware or operating system is in place on any machine that is doing the computation; networks can be heterogenous [CN01].

Load balancing techniques, though not used in high-level virtual machines, have been successfully implemented in Xen and other virtual machine systems [BDF+03].

3.3.2 Machine Swapping

Virtual machines, in addition to distributing computation, can help facilitate the distribution of users throughout the network. As true virtual machines can be migrated from system to system without the user ever noticing, virtual machines have the potential to make the operating environment as mobile as the users themselves. In most modern networks, users can log into any physical machine and see the same desktop environment published to their desktop. Computation, on the other hand, is usually tied to the particular physical machine. Using a virtual machine can guarantee that computation results will be the same, no matter what type of physical machine is being used. If the user moves, the computation environment will stay the same since the environment will be independent of the physical device. In this way, virtual machines can help users move between physical machines without penalty [CN01].

Though high-level language virtual machines do not facilitate machine swapping, this technique has been used on a number of virtual machine systems, including Xen. If a machine appears to be overloaded, the Xen system will attempt to offload a Xen process to another physical machine in order to achieve better overall throughput [BDF+03].

3.3.3 Failure Protection

If a virtual machine facilitates environment migration, then it can provide a fault tolerant mechanism and help software recover from hardware failures. When a virtual machine is used for fault protection, it is known as a hypervisor. No piece of hardware is ever perfect, and over time hardware will break. In a standard computing environment, if the CPU fails then it will bring down all processes that it is running at the time. If this happens, some critical processes may crash and data can be lost. A hypervisor prevents this form of catastrophic system failure by replicating the execution of a primary processor on other processors monitored by the virtual machine. Periodically, the hypervisor will suspend the process and synchronize results across all backup processors. If the primary processor fails at any point, one of the backups takes over. While this hypervisor system is rather inefficient - it duplicates all computation - it can withstand hardware failures that would bring down other processes [CN01].

Though this technique has also not been used in high-level language machines, it has been explored using virtual machine for user programs that must not fail, such as those for the space shuttle and other programs. A form of failure protection developed by Thomas Bressoud was about twice as slow as standard computation [BS96].

3.4 Dynamic Optimization

Virtual machines can use dynamic information to optimize code in order to improve performance. While dynamic optimization is not yet able to improve code execution to equal the speed of highly-optimized, platform-specific code, dynamic optimization techniques can vastly improve program performance. By taking advantage of runtime information and by recording code behavior, a virtual machine can selectively optimize and improve code execution speed dramatically. Virtual machines commonly perform two types of dynamic optimizations, platform independent optimizations and platform dependent optimizations.

3.4.1 Platform Independent Optimizations

Virtual machines perform several platform-independent operations, such as the propagation of constants, the hoisting of loop invariants, and the creation of superblocks. In constant propagation and loop invariant hoisting, the virtual machine is able to use its runtime information about program behavior in order to make assumptions about the likely value of a variable. This information, gathered through runtime profiling, is incorporated directly into the optimized code to allow the code to run more quickly in the general case. When formulating blocks and superblocks to optimize, the virtual machine can gather statistics about certain paths through the program code and optimize these pathways accordingly. In this case, the runtime information suggests which pathways to optimize heavily, which pathways to optimize lightly, and which pathways to ignore. These optimizations will typically double or triple code execution speed, and occasionally boost performance by an order of magnitude [SN05].

3.4.2 Platform Dependent Optimizations

A just-in-time compiler, also known as a JIT, can take advantage of features on the host system to achieve better performance. A JIT takes platform-independent bytecodes as input and compiles them to a system-specific form using a process similar to that used by a traditional compiler. In contrast to bytecode interpreters, which often

execute only one bytecode at a time, a JIT is able to optimize multiple bytecodes at once. JITs can achieve better performance by making use of hardware capabilities not specified in a virtual machine, even reordering instructions in order to take better advantage of underlying hardware. Though JITs use many techniques used by traditional compilers, such as peephole optimization and whole program optimizations, they must use extremely fast algorithms to do so, since the time spent optimizing the platform-dependent code increases execution time. JIT optimizations often speed up performance by one or two orders of magnitude, even when accounting for the overhead of running the just-in-time compiler while the program in running [SN05, Ayc03].

3.4.3 Ordering of Optimizations

Most virtual machines perform dynamic optimizations in stages. Often, the virtual machine will begin by interpreting one bytecode at a time, and profiling the executed code to build up run-time information. If a chunk of code is commonly executed, then it will be optimized using simple platform-independent techniques. If it is used even more frequently it may be optimized in stages, using iterative improvement techniques to further improve code quality. Virtual machines often will not perform extensive optimizations on the entirety of the guest program, due to the overhead of having to perform those optimizations. Additionally, heavily-optimized code generally can consume a large amount of space. There are some virtual machines, though, that will only execute code that has been compiled and will not interpret anything; these virtual machines, such as the Gambit system for Scheme, will compile the bytecode to a machine-dependent form before running any code. As time goes on, however, these virtual machines will use the JIT to recompile the bytecode, taking advantage of the increasing runtime information available [SN05, Tai98, FM90].

3.5 Security

While virtual machines provide a method for platform independence, they also provide a mechanism for enforcing a security policy. Since virtual machines operate as an intermediate layer between the user program and the operating system, a security policy implemented by a virtual machine can restrict hardware access or enforce data protection mechanisms. Virtual machines have approached this problem in four ways, via internal controls, via external access restrictions, using runtime analysis, and via program logging. Many virtual machines combine these approaches for stronger security.

3.5.1 Internal Controls

Programs that run on virtual machines often run in "sandboxes", restricted environments that isolate the program from the rest of the system. Virtual machine sandboxes typically implement a number of internal controls as well. For instance, security policies are enforced by performing checks on pointers, ensuring that all pointers are valid references. Equally important, security policies can verify data types, ensuring that data elements are not treated as references and that references are appropriately typed. Security policies can also protect a program's internal data by preventing another program from accessing its data. Lastly, the security policy must ensure that the program never interferes with the virtual machine's internal code; otherwise the security policy could be exploited or compromised entirely. If a sandboxed program attempts to violate any of these aspects of the security policy, the virtual machine can terminate it [SN05, CN01].

High level language virtual machines typically have a number of internal controls. For instance, Java has support for array bounds checking to ensure that users do not try and access invalid values. Some virtual machines support data access restrictions at a more abstract level, such as scoping rules for variables and methods.

External Access Restrictions

The virtual machine "sandbox" consists of two parts, the virtual machine core and the security manager. The virtual machine core is the part of the virtual machine that is trusted implicitly and creates a secure environment for user code. On many virtual machines, the core consists of the virtual machine's code loader and execution engine, though more advanced virtual machines may also include a suite of standard libraries as well. The security policy is used to monitor what types of files are loaded into the system. For instance, well-designed security manager should forbid programs from loading arbitrary files from the hard drive or the network; only trusted components and trusted programs should be allowed to do so. More elaborate virtual machines may have more sophisticated security policies that will allow for specific programs to have more fine-tuned control rather than blanket loading privileges. Anything that is not in the trusted core is usually examined by a verifier, a virtual machine component that ensures that the loaded code is statically safe or has appropriate runtime checks [SN05].

For many years, high-level language virtual machines provided few restrictions on ways to access outside resources. With the advent of the Internet, however, this sort of monitoring became essential in order to protect a computer from malicious attacks

by imported code. The Java Virtual Machine probably provides the most complete security policy of any high-level system.

3.5.2 Monitoring

Some high-level language virtual machines have the ability to run code that is untrusted, programs that cannot be verified by a virtual machine's security manager. Typically, these programs have pointers, casts, or indirect loads that do not follow standard techniques. Though untrusted programs may be perfectly safe, they cannot be statically checked and therefore cannot be guaranteed. In order to allow a program to run untrusted code, the user must explicitly relax the controls on the runtime environment, allowing the virtual machine to load and execute the code [SN05]

The ability to run untrusted code is not a very common property, mostly because security is so important. In high-level language virtual machines, this property is only found in the C# virtual machine [NR302a, NR302b].

3.5.3 Logging

Virtual machines are also capable of logging activity occurring on the virtual machine, allowing administrators or other qualified users to understand how a security breach occurred. Though many programs log events for security reasons, attackers who manage to compromise the program can also compromise the logging scheme. By doing the logging within the virtual machine, however, the record keeping is more securely implemented. If the program crashes, the virtual machine will be able to continue to log events. If enough information is logged when the program is executing, it may be able to reconstruct the execution of a virtual system and replay the process by which a system was attacked. While this is especially important in system virtual machines, which host entire operating systems as well as guest programs, logging remains an important though less explored service in high-level language machines [CN01].

As of yet, high-level language virtual machines do not implement logging, possibly due to the additional performance overhead that logging may entail. Logging is typically used by virtual machine monitors such as Xen [CN01].

3.6 Development

The standardization provided by a platform-independent system can significantly simplify the task of programming by decreasing the amount of code that needs to

be written and increasing project development speed. Virtual machines cut down on the volume of code created by allowing the designer to target a single instruction set. If the code were designed for many conventional systems, specialized code would be necessary to handle each specific operating system and potentially each specific hardware architecture. Code designed for virtual systems need only have one version, since the guest running on the virtual machine need not know anything about its host. Because the amount of code that needs to be developed is potentially smaller, project design can proceed much more quickly. Since a significant percentage of total project development time can be spent spent tracking down porting bugs, languages that run on virtual machines often do not incur this overhead. Targeting a virtual machines can often increase code development speed since a development team need not focus on code portability [SN05].

Chapter 4

The ++VM

The remainder of this work presents the ++VM, a high-level language virtual machine that highlights features described in previous chapters. The ++VM is designed to combine the best low-level features found in multiple languages. One of the major design goals of the ++VM is the creation of a flexible instruction set which integrates easily with multiple high-level languages. This chapter gives a summary of the features available in the ++VM, including tag support, memory management, object support, opcode classes, and annotations.

4.1 Overview

Figure 4.1 shows a general overview of the links between the different components of the ++VM, highlighting many of the features that will be discussed in later chapters.

4.2 Tags

The ++VM has fourteen different tags: seven data tags and seven reference tags. These tags are stored in parallel with the data using tagged memory. For more information about the tagging system, see Chapter 7 and Appendix A.

4.2.1 Data Tags

The ++VM supports seven data tags. Five of these are used to represent integers with widths from 8 to 128 bits, and two of these tags are used to store floating point values. One data tag is reserved for future extensions.

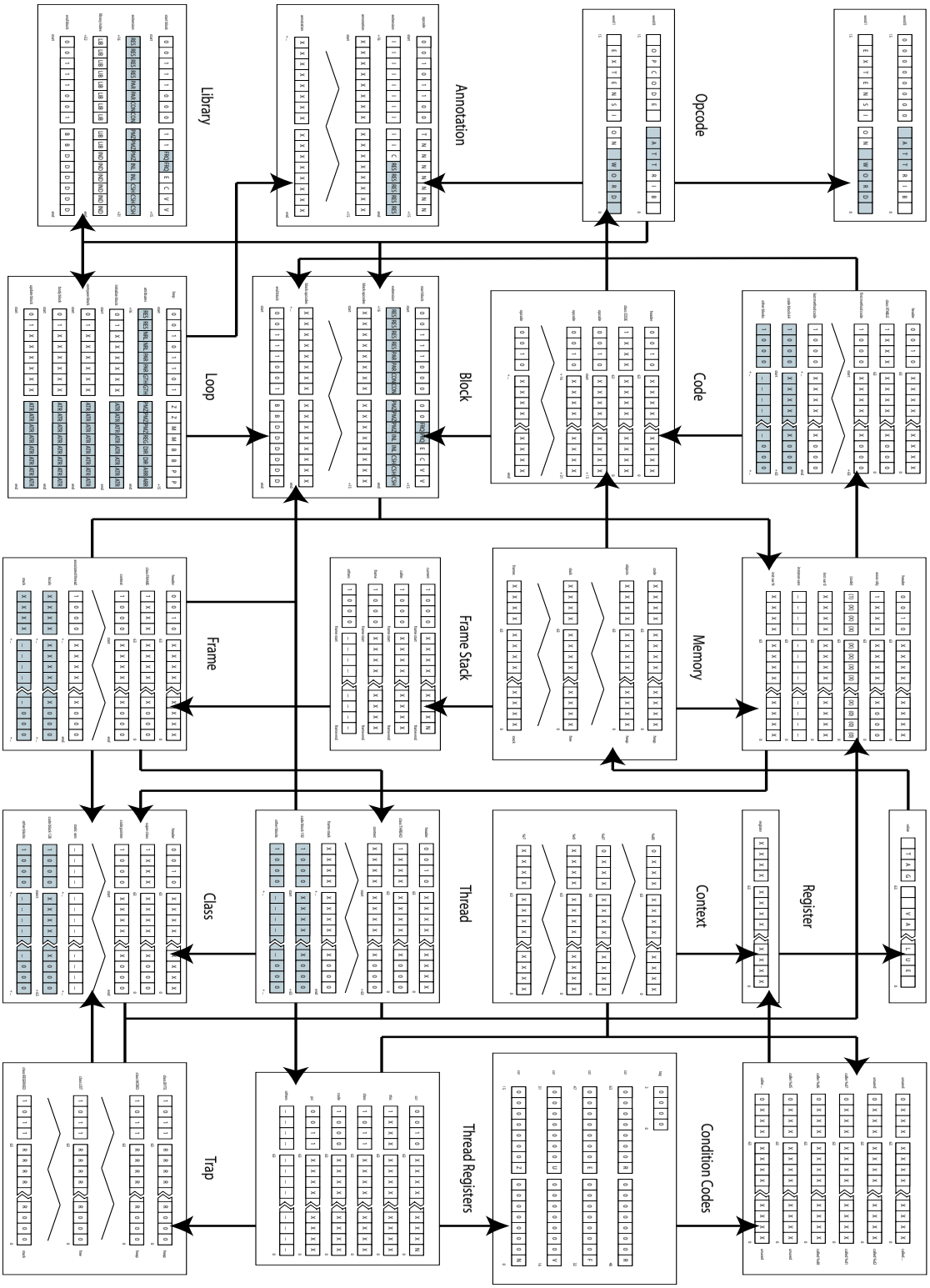

Figure 4.1: The ++VM

Byte A byte is an unsigned 8-bit integer. This is large enough to store one ASCII character. Memory in the ++VM is byte addressable.

Short A short is an unsigned 16-bit integer. This is large enough to store one Unicode character.

Int An int is a signed 32-bit integer.

Long A long is a signed 64-bit integer. This is the preferred computational unit in the ++VM, and is also the register size.

Ultra An ultra is a signed 128-bit integer. Ultras are stored in little-endian format using two memory locations or registers, and must be aligned on a 16-byte boundary.

Float A float is a signed 32-bit floating point number, IEEE 754 standard.

Double A double is a signed 64-bit floating point number, IEEE 754 standard.

4.2.2 Reference Tags

The ++VM has seven reference tags. Six of these tags are object tags, references to collections of data containing a number of header fields in addition to the object information. The remaining tag is a pointer tag, a general pointer to any memory address. For more information about tags, see Chapter 7 or Appendix A; for more information about objects, see Chapter 5.

Object An object reference is a 64-bit reference to a long-aligned object. The associated object pointer of this object points to the object's class, as is common in many object-oriented languages. The object does not have a code pointer. This tag is the generic object which will be the workhorse of the ++VM virtual machine. It is the default tag created when objects are allocated.

Object with Code An object with code reference is a 64-bit reference to a long-aligned object. The associated object pointer refers to the object's class. The object has a code pointer which points to methods which have been just-in-time compiled for this particular class instance. This tag can be used when a complicated object will be accessed frequently but modified rarely, and thus is a good candidate for constant folding and other object-specific optimizations.

List Element A list element reference is a 64-bit reference to a long-aligned, Lisp-like object. The associated object pointer points to either an Object or an other List Element, not to a class. The class for a list element can be retrieved by checking a per-thread trap vector. List elements do not have a code pointer. A list element can be used to minimize the amount of representation overhead needed for storing list structures.

Class A class reference is a 64-bit reference to a long-aligned class. The associated object pointer of this class points to the class's superclass. The code pointer of this class points to the methods of the class. A class contains the static variables and generic code available to its instances.

Future Object A future object reference is a 64-bit reference to a long-aligned object which awaits computation. The associated object pointer of this object points to the object's class, as is common in many object-oriented languages. The object does not have a code pointer. Future objects and lazy evaluation may be helpful in multithreaded environments.

Future Object with Code A future object with code reference is a 64-bit reference to a long-aligned object that has not yet been computed. The associated object pointer points to the object's class. Additionally, the object has a code pointer which points to methods which have been just-in-time compiled for this particular class instance. This object tag is identical to the future object except for the additional code pointer.

Pointer A pointer is an arbitrary 64-bit pointer to any location in memory. The values to which pointers point do not have to be long-aligned. A pointer can be used to perform operations which might not be safe for garbage collection, but the objects on which these actions are performed should be pinned in place to avoid invalid memory references after garbage collection.

4.3 Memory Management

There are three major components to the ++VM memory management system. For intermediate values, it uses a series of registers and a stack. For more permanent values, the machine uses either an object-oriented or pointer-based storage technique. The virtual machine also provides a number of support registers, called machine registers, which facilitate machine execution. For more detailed information about memory management, see Chapter 6.

4.3.1 Intermediate Values

The ++VM has sixteen registers to store intermediate values, divided into two groups. Eight registers are data registers, and are used to hold and manipulate data elements directly. The remaining eight registers are reference registers, and are used to manipulate reference values. Since there may not be enough registers to hold all intermediate computation values, the virtual machine also provides a stack. All registers, and all stack elements, are tagged.

4.3.2 Additional Storage Areas

There are two main ways to access data in the ++VM, via a field offset from an object or via a pointer. The primary way to access memory in the ++VM is via an object offset. As components in the system can be represented using objects, the object offset notation allows easy access to memory locations. The pointer reference tag allows the ++VM to access arbitrary memory locations, potentially including object header fields. All memory locations are tagged.

4.3.3 Machine registers

The ++VM has a number of registers which store information about the internal state of the virtual machine. These include registers for keeping track of the current object, class, frame, and thread, as well as information about executing code. Additionally, the ++VM uses a condition code register to store result state from the last instruction that was executed. This register is used for conditional branch tests.

4.4 Objects

Objects are essential to the functioning of the ++VM All components in the ++VM system can be described using the object format. For more information about objects, see Chapter 5.

4.4.1 Header

Every object requires a 64-bit header in order to store information associated with that object. The first part of this field contains information such as object size, object lock information, and various bits for the garbage collector to use. The second

part is an associated object pointer, which can point to an object's class or a class's superclass.

4.4.2 Code

Some objects have code associated with them, which can be accessed via an object's code pointer. For instance, all class objects have access to the code which is needed by the instances of that class. However, some object instances may also have code associated with them as well.

4.4.3 Attributes

Instructions that create or modify objects can use attribute bits to specify more information about how that object can be used efficiently. Attributes can include a wide range of information, from the expected lifetime of a particular object to the kinds objects that will interact with a particular object.

4.5 Opcodes

Every opcode in the ++VM is a series of 16-bit short words, each containing a mixture of required and optional instruction information. The first byte of the first opcode word specifies an operation family, such as math addition or object creation. The remaining byte, and zero or more additional instruction words, will contain more specific information about what kind of instruction to execute. The virtual machine defines 64 opcodes in several broad categories, including math operations, memory operations, control operations, creation operations, thread operations, annotation operations, and miscellaneous operations.

Languages implemented on the ++VM are unlikely to use all of the opcodes provided by this language specification; instead, most will use some subset of the opcodes specified by the virtual machine. Some languages will not allow for the execution of certain opcodes, while others will ignore certain attributes of other instructions. For instance, some of the opcodes specified may be potentially unsafe. Certain operations, such as pointer arithmetic, are difficult or impossible to verify before the code is executed. An opcode verifier can be used to ensure that some of the opcodes are safe; however, certain opcode attributes or even certain code constructs cannot be verified. While the ++VM will support these operations, certain language preferences may not permit these operations.

For more information about opcodes, see Chapter 8.

4.6 Attributes & Annotations

Some of the bits in ++VM instructions, as well as some ++VM opcodes themselves, are meta instructions. These attributes and annotations provide a more detailed picture of the code's high-level properties and can be used to provide situational information to the runtime system. Using attributes and annotations may allow the virtual machine to make more informed decisions about memory management and code generation, resulting in performance improvements. The ++VM also allows implementations to define their own custom annotations. These annotations can then be used by ++VM compilers to extend the ++VM system, making it simultaneously more flexible and more powerful. For more information about annotations, see Chapter 10.

Chapter 5

Object and Object Attribute Support

Over the last ten years, object-oriented programming has become a common way of writing code. Object-oriented languages work by manipulating hidden structures via messages, and they take advantage of data abstraction and encapsulation in order to treat all types of data in a uniform manner [Mit03]. Many virtual machine languages have adopted the object-oriented paradigm, using objects as the basic operational data structure in a language. The object layout, in addition to defining a number of properties of the underlying virtual machine, also has a significant impact on the underlying performance of the host machine.

There are several differences between objects in the ++VM and objects in other high-level language virtual machines. First, in contrast to some object-oriented languages, the ++VM specifies the internal layout of objects in the system. While this design decision imposes a number of constraints upon the language implementation, it allows for a uniform treatment of objects across systems and languages. Additionally, objects in the ++VM can be influenced by attributes in the opcode stream. These attributes can provide more information to the system about an object's lifetime and behavior.

The remainder of this chapter specifies the object layout, considers a number of object attributes, and ends with a discussion of the benefits of these features for the ++VM.

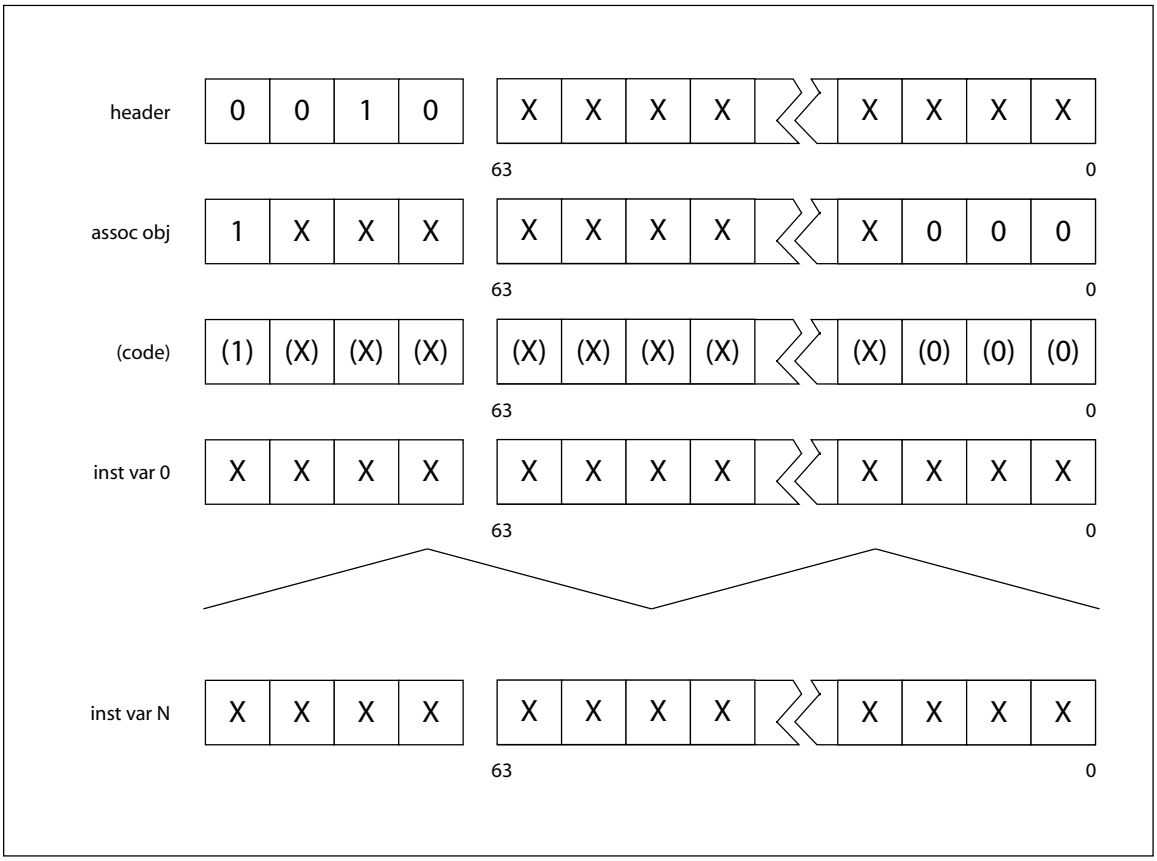

Figure 5.1: A ++VM object

5.1 Object Layout

Figure 5.1 depicts the layout of an object in memory.

5.1.1 General Objects

The object layout is extremely important for the ++VM. The first part of the object, consists of two or three parts: the internal information field, the associated object pointer, and an optional code pointer. Following these fields are the object instance variables. The instance variable store the object-specific information and hold the program data.

Internal Information Field The internal information field is typed as a short, and contains four 16-bit fields.

The highest order short is used to hold the number of words that are needed to store the instance variables of this object. This field can be used to calculate the size of the object, as the header size is a constant. The field size can be read by the program, but should not be modified. The value of this field is retrieved when the one-operation instruction "field" is executed.

The next short in the internal information field is used to hold a lock. If the short value is zero, then the object is not locked. If the value of the short is less than 256, then the short value holds the number of times that the lock is held. If the value of the short is greater than 256, then the short value serves as an index into a lock table and can be used to retrieve a more complicated lock structure.

The two low order shorts are reserved for use by the garbage collector and future extensions of the virtual machine. Some of these bits will be used to determine whether an object has been examined by a garbage collector. Other bits may be used by the thread scheduler or other internal mechanisms.

Associated Object Pointer The second part of the header is a pointer to an associated object. This will vary depending upon the type of the specific object. An overview of the six object types can be found in Section 4.2.2.

If the object is a standard object, then the associated object pointer references the class associated with that object. The class will contain the method vtables as well as the static variables which can be accessed by that object.

If the object is an object with code, then the associated object pointer also references the class associated with that object. The class will contain the generic method vtables as well as the static variables which can be accessed by that object.

If the object is a class, then the associated object pointer references the direct superclass of that class. Even though classes themselves are objects, the associated object of a class object does not point to a meta-object class.

If the object is a list element, then the associated object pointer of the list element type is a reference to one of two types of objects. If a list element's associated object pointer is a reference to a standard object, it can be thought of as a Lisp-like atom. On the other hand, if the list element points to another list element, the list element can be thought of as a Lisp-like cons cell. The class for the list element can be retrieved using the trap vector; for more information on trappnig, see Section 9.3.2.

Code Pointer The third header component is a code pointer. The code pointer is a reference to code perhaps just-in-time compiled for this particular object. The only types of objects which have code pointers are objects with code and class objects; all other types of objects must follow their associated object pointers in order to find an object with a code pointer. If an object does not have a code pointer, then that object's header can be shortened.

Instance Variables Following the object header are the object's instance variables. Instance variables of the class are stored in memory addresses above the code pointer (or, if the code pointer is not present, after the associated object pointer). The number of instance variables is determined by the object's length.

5.1.2 Specific Objects

Some objects in the ++VM are more complicated than the standard object. Though these objects have the same memory layout as other objects, their properties and behavior are more complex. These objects include classes, arrays, and lists.

Class Objects Classes in the ++VM are objects, and the virtual machine can treat them in the same way as it treats other objects. In memory, classes are distinguished by the class reference type. The header of a class object has been described as above: the associated object pointer of the class will point to the class's superclass, and all classes will have code pointers. The remaining fields in the class object will contain the class's static variables.

Arrays The ++VM dose not distinguish between arrays and objects at the type level. Arrays are treated identically to any other object, except that arrays are objects with lengths determined at runtime. In contrast to a standard object, whose size is determined statically via its superclass, the size of an array is determined from the length parameter to the array constructor. When an array is allocated, the length of the array is stored in the length field of the object header.

At runtime, the only possible difference between an array and an object is that the array indices may need to be bounds checked. The need for bounds checking is controlled by the object's tag bits: if the bounds check bit is set, then the length of the object is compared with the requested index. Not all arrays need to be bounds checked, as an object attribute can be used to disable explicit bounds checking.

Lists The ++VM also provides native support for Lisp-like cons elements through the list element reference type. List elements consist of the header information followed by two object references.

If the object is a list element, then the associated object pointer references another object. All list elements must be of the same class type. Using the associated object pointer in this way allows the ++VM to minimize the amount of space overhead necessary to store list elements by storing the list element class only nce in the per-thread trap vector.

5.2 Object Behavior

Allocation All objects in the ++VM are dynamically allocated. Every ++VM object must be allocated on a 64-bit boundary. Objects can be allocated in the stack or the heap; the location of allocation is controlled by an object attribute. Stack allocated objects do not have to be garbage collected, as they are allocated directly into a method's stack frame. Additionally, objects can be allocated in specific cache lines. The memory allocator can take a similar object as a parameter and to allocate the new object in a different cache line than the passed parameter. Using this allocation scheme, it will be possible to put commonly-used objects in different cache lines to minimize the number of cache misses. For more information, on object allocation, see Chapter 10.

Even before allocation, objects can be influenced by other, more general object attributes. The frequency of object access, as well as the length of the object lifetime, can be specified as attributes. Another attribute can specify whether this object will ever be null and can be used to possibly eliminate null checks on a particular object.

Referencing An object reference points to the associated object field of the object. In order to access values in the information field, the virtual machine will have to subtract the appropriate number of shorts to access the data.

When the virtual machine searches for code associated with an object, the tag bits are used to check whether that object actually has a code pointers. If the tag bits indicate that a code pointer is present, the virtual machine then executes the object-specific code. If the tag bits indicate that no code pointer is present, then the virtual machine must look for code via the associated object pointer.

The first object instance variable is located one or two long words beyond the location of the object reference. Object instance fields can be of any type, either data type or reference type, though the types of these fields are specified by the object's

class. Since memory is tagged by the long word, data of the same type can be packed into the same long word: eight bytes can be stored together in the same long word as opposed to having to store eight bytes in eight different long words.

Tagging All objects in the system have the most significant tag bit set, marking them as one of the six reference types as described in Section 4.2.2. The remaining bits control different properties of the object, such as whether or not it has code. The tag bits are not the same thing as the object's class type; the class type must be determined by following the object's associated object pointer. The tagging bits, in conjunction with the length field, control hde extent of the object should be examined by a garbage collector.

Auto-boxing In the ++VM, primitive data types are stored as primitives. Primitive data types are not stored in special classes when they must be treated as reference types in arrays. In order to do this, it is necessary to perform an extra check prior to method invocation. Before a non-static method is executed, the type of the this pointer, %tc, is examined. If the tag bits of the this pointer indicate that the register value is a data value, then the method is trapped by the virtual machine before it begins to execute. A class lookup can then be performed using the tag bits as a class index. In this way, the tag bits serve as an indirect form of a class pointer. This system allows primitive data types to be the targets of methods in the virtual machine while avoiding the slowdown of an additional levels of indirection; in many method invocations, the JIT compiler should be able to omit the tag checks when primitive data types cannot possibly be the target objects. For more detailed information about trapping, see Section 9.3.2.

Garbage Collection All heap objects in the ++VM are garbage collected. The behavior of the garbage collector is unspecified, although generational garbage collection may be facilitated by the life-length object attribute. Since objects must be allocated on 64-byte boundaries, the three least significant bits in each object reference are available for the garbage collector to use. All references, including the associated object pointer and code pointers in the object headers, must reserve these bits.

5.3 Object Attributes

When created, objects can have a number of attributes. An additional instruction, the attribute instruction, is able to change these attributes at later stages of program execution in order to provide more accurate information to the virtual machine.

Access Frequency The access frequency attribute uses two bits to determine how frequently an object is being accessed. If an object is frequently accessed, this attribute should be set appropriately so that the just-in-time compiler keeps this object in the cache as much as possible. Alternatively, if the object is accessed infrequently, the object should not be cached.

Life Length The life length attribute uses two bits to determine whether the object will last for three lengths of time, or whether no generational information can be determined. Some of the objects created by the ++VM may be only temporary wrappers for data, while objects will be nearly permanent fixtures of the virtual machine. If the virtual machine can capture this information, it may be able to allocate new objects in an appropriate garbage collector generation, improving garbage collection speed.

Pinning Since the ++VM allows pointers to objects, the garbage collector must not move objects which are referenced by pointers. This attribute prevents the garbage collector from garbage collecting or moving an object, ensuring that pointers to this object will always remain valid. The ++VM may choose to place pinned objects in a separate memory section to facilitate garbage collection for other objects; however, this is not required and pinned objects can be intermixed with unpinned ones.

Garbage Collection The garbage collection attribute uses two bits to fine-tune the garbage collector for this particular object. This attribute can be used to enable or disable garbage collection on a particular object, though the virtual machine makes no guarantee that the object will not be moved around even when garbage collection is disabled. One setting of these bits will run the garbage collector immediately on the object, allowing this attribute to function as a freeing mechanism.

Optional Null Checks This attribute will instruct the just-in-time compiler to create or remove null checks in the native code that it creates. Though disabling null checks may make virtual machine instructions potentially unsafe, it has the potential

to greatly increase the execution speed of the program. This attribute can be of use in multiple types of languages. First, not all object-oriented languages require null checking, and this attribute can be used to achieve that behavior. Additionally, even in object-oriented languages that require null checking, this attribute may have uses. The programmer might be able to determine that an object will contain data, even if the compiler is not able to do so, and allowing a hinting mechanism may lead to faster performance on trusted libraries.

Stack Allocation Although ++VM objects are typically allocated on the heap, this attribute allows the virtual machine to allocate an object on the stack. Stack-allocated objects are potentially more efficient, as they do not have to be moved around by the garbage collector. Instead, stack-allocated objects are automatically garbage collected when a method terminates: they are destroyed along with their associated stack frame. Stack-allocated objects, however, are potentially unsafe, as a stack-allocated object may be destroyed while it is still referenced by a heap allocated one. If a compiler is able to determine that invalid references will not occur, or if the implemented language allows unsafe behavior, this attribute can be used to achieve faster performance. Additionally, since stacks are objects, this allocation process allows for objects to be allocated within other objects. Objects allocated on the stack should be pinned on allocation so they are not moved into the heap.

Cache Lines The ++VM allocator attempts to allocate objects in different cache lines to maximize cache effectiveness. When an object is created in the ++VM, another object can be passed to the allocator as an object attribute. The allocator will put the new object in a different cache line than the other object; if the two objects are then used together, the two may not conflict in the cache. Previous papers on this type of cache-conscious allocators have performance improvements of up to 42% over standard allocators [CHL00].

5.4 Features of Objects

The ++VM object specification calls for a number of features that are not often found in other virtual machines. These attributes are designed to add flexibility and expressive power to the system. By giving more information to the virtual machine, the compiler and the coder may be able to take advantage of this in a variety of ways.

General Objects In the ++VM, every component of the system is an object. This means that all objects can be treated in the same fashion, whether they are a user objects or system components. Though class objects must be handled specially, as their associated object pointer points to a superclass as opposed to the class Class, they share the same general layout as everything else. This consistent framework is a more elegant way to design a virtual machine, and is not found in many other systems. Additionally, this will create a simpler sequence of hardware instructions because component properties will not have to deal with different cases for different types of instructions.

Objects as Primitives Objects are an essential part of the ++VM virtual machine. The ++VM includes support for six primitive reference types, as described in Section 4.2.2. In contrast, most hardware architectures do not directly support objects, as objects are not primitive types in many machines. By providing direct support for objects, the ++VM attempts to be a middle layer between virtual machines implemented purely in software and real machines implemented completely in hardware. The objects are still platform-independent, as required by the virtual machine.

Standardized Layout Every object in an object-oriented language is associated with a class. In many current virtual machines, the location of this pointer is only loosely specified. In the ++VM, however, the class pointer, code pointer, and lock must immediately precede the object data for two important reasons. First, the ++VM allows general pointer types, which must be able to figure out where object fields are located. If the object layout is different between different implementations of the ++VM, platform-independent pointer manipulation may not be correct. In order for this virtual machine to support platform-independent pointer arithmetic, objects must have the same layout. Additionally, specifying the location of the class pointer can provide a number of hints to the JIT. If the JIT knows general properties of all objects, it can be used to improve overall system performance. By keeping the class pointer, code pointer, and lock in the same place, it may be possible to standardize certain JIT optimizations across multiple implementations of the virtual machine.

Code Pointers Using code pointers can allow for faster performance for specific objects as well as logical code partitioning. Code pointers can improve performance by allowing per-object code optimizations. If an object has many near-constant fields,

the virtual machine may be able optimize them just-in-time. In most systems, the optimized code is appended or inserted into the generic code, meaning that runtime checks may be necessary to determine whether the optimized or the unoptimized code should be used. Additionally, using code pointers would have the additional benefit of logically grouping similar snippets of code in memory. In many JIT implementations, all optimizations are performed to a basic block in memory, with guards in place to prevent code from going down the wrong pathway; though the optimizations themselves are separated into blocks, these blocks can be linked into pathways if they are logically related.

Object Attributes Objects in the ++VM have a variety of attributes, as described in Section 5.3. These attributes help make the virtual machine much more flexible than standard virtual machines. Many of these attributes also improve virtual machine performance, such as attributes that suggest caching strategies or attributes that provide information about an object's lifetime. While some of these attributes may be disabled by a particular language, attributes provide language designers with additional means of providing information about an object's behavior.

Object-Oriented Registers The ++VM provides object-oriented registers, something that today's hardware does not provide. Many high-level languages use objects, pointers to complex data structures; current hardware largely deals with uninterpretable pointers. By defining object-oriented registers, the ++VM virtual machine will be able to provide assistance to both the hardware and the software running on the virtual machine. Even if the object-oriented registers in the ++VM may not provide a performance boost on current hardware, these registers will become more efficient as hardware begins to provide better support for objects.

Auto-boxing The ++VM supports auto-boxing, allowing numeric primitives to be used without an extra level of indirection. This makes it possible to use numeric primitives in places where other languages would require wrapper classes. This saves space, as it is no longer necessary to use additional memory to hold the wrappers; equally importantly, it saves the programmer from having to explicitly remember to wrap the object. While it may add additional levels of complexity and may require some runtime checks, the compiler may be able to determine where these checks can be eliminated and mitigate any performance penalty.

Viewable Object Size In the ++VM, it is possible for the program to access the field that contains the object size. This approach is not common in many dynamically-allocated memory systems: past virtual machines have hidden the actual object size. There are three major advantages to allowing the object size public information. First, the virtual machine is able to have an accurate understanding of the meaning of every bit in memory; it is not necessary for implementations to hide memory bits from the virtual machine. Additionally, since the pointer data type makes it possible for the program to access any location in memory, exposing the object size makes it possible to allow virtual machines to map virtual machine memory directly onto main memory (though it makes it possible for careless or malicious programmers to change an object size). Lastly, it avoids the necessity of storing the length value twice for an array, as arrays must hold on to their length values for bounds checking.

Chapter 6

Intermediate Value Support

In virtual machines as well as real machines, a significant percentage of the computation process consists of moving intermediate values from one memory location to another. The ++VM attempts to minimize the frequency and the complexity of this value swapping, and does this in two ways. First, it uses a register-based system to indicate the relative importance of each of these intermediate values and to reduce the number of instructions necessary for swapping them around. Additionally, it uses a flexible memory system in order to support a variety of object-oriented and pointer-based memory operations.

This chapter first focuses on short-lived intermediate values, those held in registers on a per-method basis. It introduces the register system as implemented in the ++VM, explains the kinds of virtual registers needed in the machine, and discusses the advantages of this approach. Afterwards, this chapter turns to a discussion of more persistent values, such as object fields and main memory.

6.1 Registers

The ++VM machine is a register-based machine, and most ++VM opcodes are optimized for execution on registers. Registers in the ++VM can be divided into two groups, frame registers and thread registers. Frame registers are used to hold temporary values from the user code. Thread registers are used to store information about the state of the virtual machine. Both frame registers and thread registers have the same format.

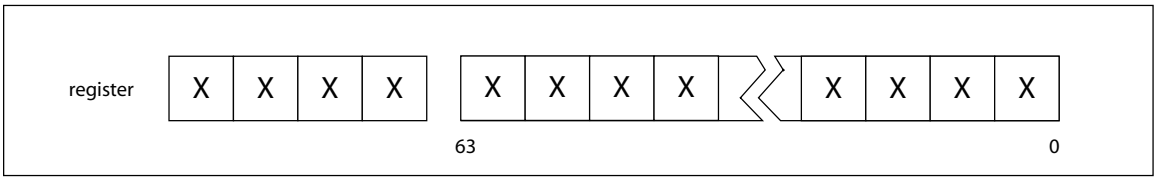

Figure 6.1: A ++VM register

Register Format As can be seen in figure 6.1, all registers hold 64 bits worth of information. Registers are tagged with an additional four bits of type information just like all other memory elements in the ++VM. Data registers are tagged with the appropriate data type, reference registers are tagged with the appropriate reference type, and machine registers are statically typed. For more information on memory tags, see Chapter 7.

6.1.1 Frame Registers

The ++VM requires a number of registers to keep track of a method's internal state. Frame registers, the registers that hold this information, are allocated on a per-frame basis. These type of registers are called frame registers because each frame will have its own set of thread registers and these registers are freed when the method ends.

The ++VM has sixteen frame registers, eight data registers, numbered %d0 to %d7, and eight reference registers, numbered %r0 to %r7. An example of a register frame can be seen in figure 6.2.

Data Registers Data registers are designed to hold data that is will be manipulated directly by the virtual machine, such as ints or doubles. All of the data registers in the ++VM are identical to each other, and must be able to store data in any of the data forms described in Section 4.2.1. Registers are typed, and the types are held in the condition code register described below in Section 6.1.3. Only even-numbered registers can hold 128-bit ultras; the corresponding odd-numbered register is used to store the lower 64 bits. When an ultra value is stored in two registers, then it is illegal to access any one constituent register as a different data type. Certain operations that take three operands, such as inner product, must select one of their operands from the low-order registers. In order to treat information in a data register as a pointer, it is necessary to explicitly cast it into a reference register. Objects in data registers cannot be directly dereferenced. For this reason, data registers are ignored during garbage collectors, as they cannot contain interpretable pointers or object references.

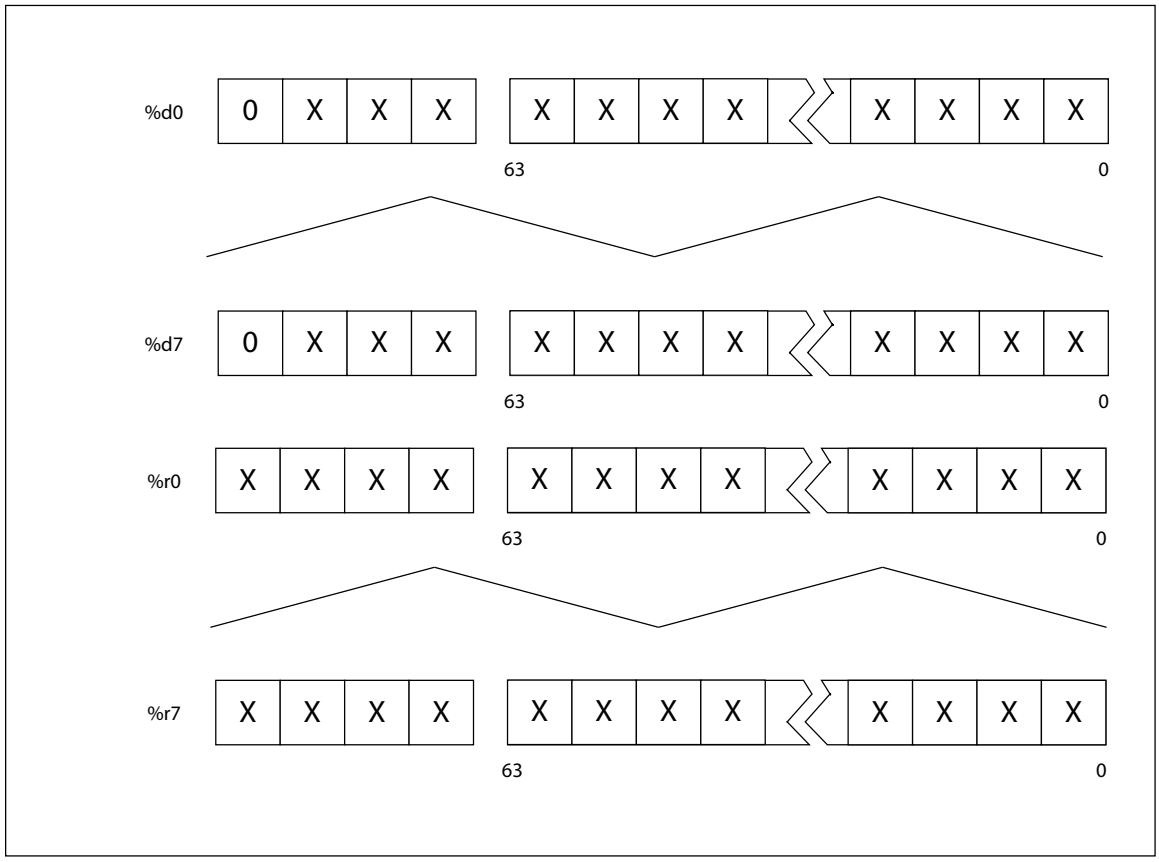

Figure 6.2: ++VM frame registers

Certain languages may still find this property a safety concern and will choose to disable this virtual machine feature.

Reference Registers Reference registers in the ++VM are designed to hold data that will be indirectly manipulated by the virtual machine, such as objects or arrays. All of the reference registers are identical to each other, and must be able to store references in any of the reference form of Section 4.2.2. None of the reference registers of the frame are used to store stack pointers, frame pointers, or other machine-specific information; this information is stored in the thread registers. Reference registers form part of the root set for garbage collection purposes. It is not possible to perform pointer arithmetic on a reference register. In order to manipulate a pointer, it is necessary to convert it to a data type using a data register, perform the desired

manipulation in that data register, then recast it to a pointer. Certain languages implemented by the ++VM may not permit these operations.

6.1.2 Thread Registers

The ++VM also requires a number of registers to keep track of the virtual machine's internal state. Thread registers, the registers which hold this bookkeeping information, are allocated on a per-thread basis. These type of registers are called thread registers because each thread has its own set of registers that must be saved when the thread is suspended. A diagram of a thread register is show in figure 6.3.

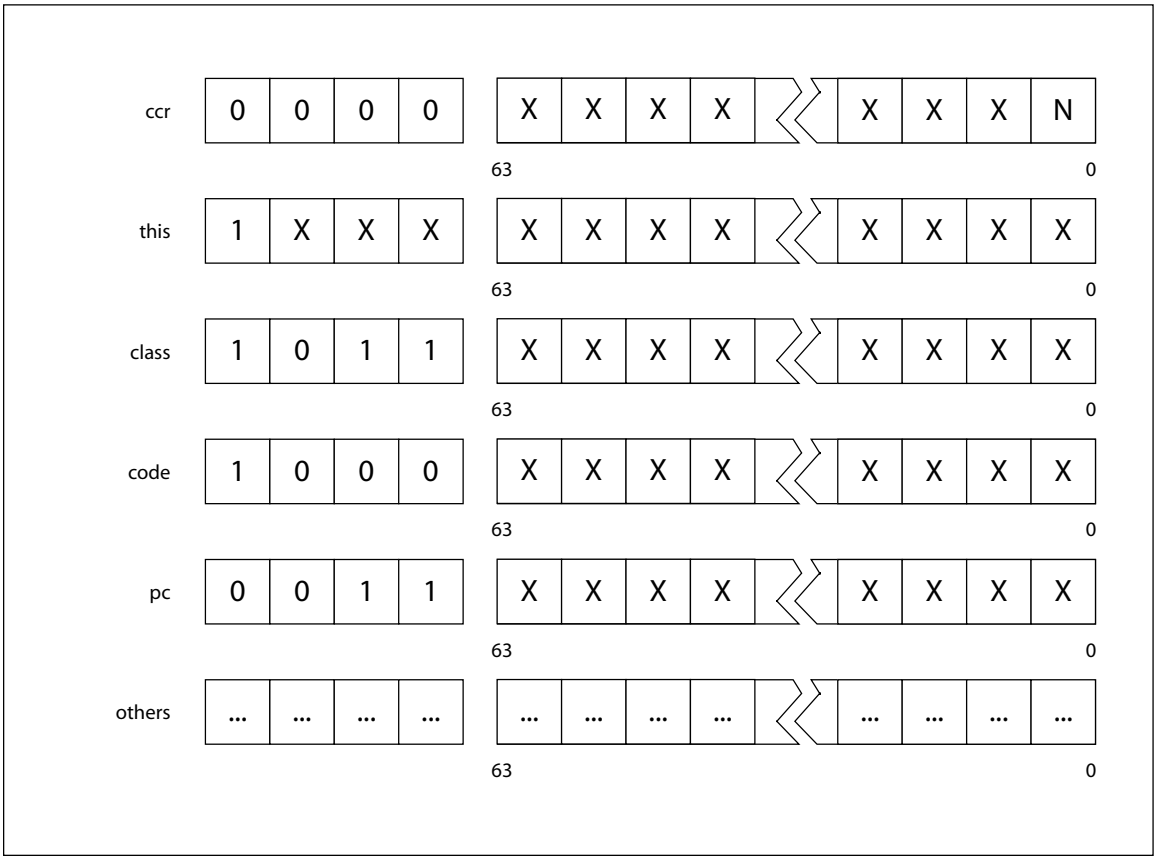

Figure 6.3: ++VM thread registers

Generally, the thread registers contain critical information about the state of the virtual machine. The contents of these registers cannot be read or changed by a

program, but code operations will set or change their values indirectly.

Condition Code Register This register holds information about the result of the last instruction. The condition code register is tagged as a collection of bytes. For more information about the condition code register, see Section 6.1.3.

This Object Register The this object register, %to, holds the object that is currently being manipulated. For static methods, it is set to zero. The this object register is tagged with the tag of the object to reference in memory; usually, the this object register will have a reference tag but it can be a data tag when auto-boxing is in use. For information about how methods manipulate the this object register, see Chapter 9.

This Class Register The this class register, %tc, holds the class of the object in the this pointer. For static methods, it is set to the class on which the method is being called. The this class register is tagged as a reference to a class. For auto-boxed values, the value of the %tc register is derived from the trap vector. For information about how methods manipulate the this class register, see Chapter 9.

Code Register The code register, %cr, holds a pointer to the current code object. The code object is an array of shorts that contain instructions for the program to execute. The code register is tagged as a reference to an object.

Program Counter The program counter register, %pc, holds the current program counter. The program counter is an index into the code object. The program counter register is tagged as a long.

Trap Register The trap register, %tr, holds the thread's trap vector. The trap vector register is tagged as an object reference. For more information about trapping and auto-boxing, see Chapter 5.

Exception Register The exception register, %xr holds the active exception handler if a stack cutting scheme is in use. The exception handler register is tagged as an object reference. For more information about exception handling, see Section 12.5.

Stack Pointer The stack pointer register, %sp, holds the stack pointer, an index into a method's stack. The stack pointer is tagged as a long.

Frame Register The frame register, %fr, holds a pointer to the current frame. The method stack, if needed, is held at an offset within the frame. The frame pointer is tagged as an object with code reference.

Machine Register The machine register, %mr, holds a reference to the virtual machine, and can be used to determine the next thread to schedule. The machine register is tagged as an object with code reference.

6.1.3 The Condition Code Register

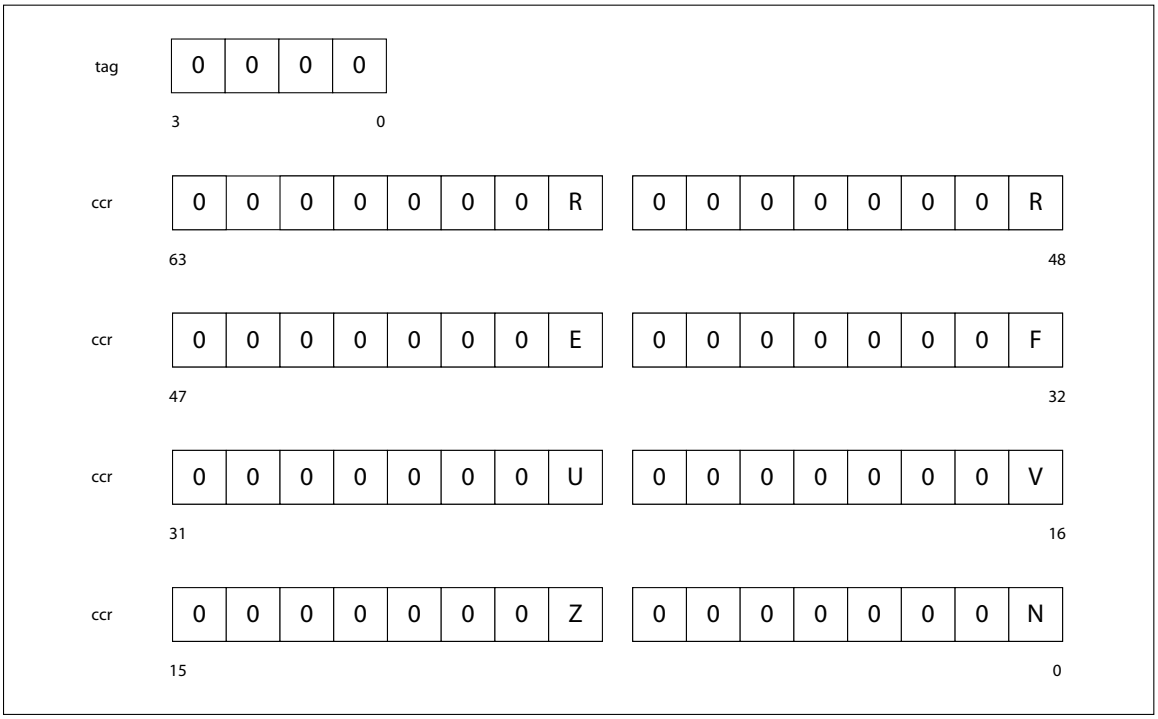

Figure 6.4: The condition code register

The condition code register extremely important to the functioning of the ++VM. This register, detailed in figure 6.4, holds eight bytes. One bit of each byte is used to encode condition and branching information. Of the eight available bytes, only six are in use.

Negative Bit The negative bit, N, is set if the result from the instruction was a negative number. Otherwise, it is clear.

Zero Bit The zero bit, Z, is set if the result from the last instruction was zero. Otherwise, it is clear.

Overflow Bit The overflow bit, V, is set if the result from the last instruction was too large for its data type. Otherwise, it is clear.

Underflow Bit The floating-point underflow bit, U, is set if the result from the last instruction was too small for a data type. Otherwise, it is clear.

Future Bit The future bit, F, is set if the result from the last instruction was a future and that future still awaits computation. Otherwise, it is clear.

Exception Bit The exception bit, E, is set if stack cutting is used to handle exceptions. It is cleared if stack unwinding is in use. For more information on exception handling, see Section 12.5.

Reserved Bits The remaining bits are reserved.

6.2 Register Windows

In the ++VM, a register window is used to hold the current frame registers. Register windows minimize the amount of memory that must be moved internally and provide a convenient manner of passing parameters to methods. A pictorial description of a register window can be seen in figure 6.5

6.2.1 Description

A register window can be thought of as a contiguous subset of all available registers. Of all of the data registers and reference registers available to the virtual machine, a method is only able to access those registers which are within the window. The register name %d0 become relative offsets within the collection of all available registers: when a new method is called, the window shifts and %d0 points to a free registers; when the method returns the window shifts back and the old registers return to their original

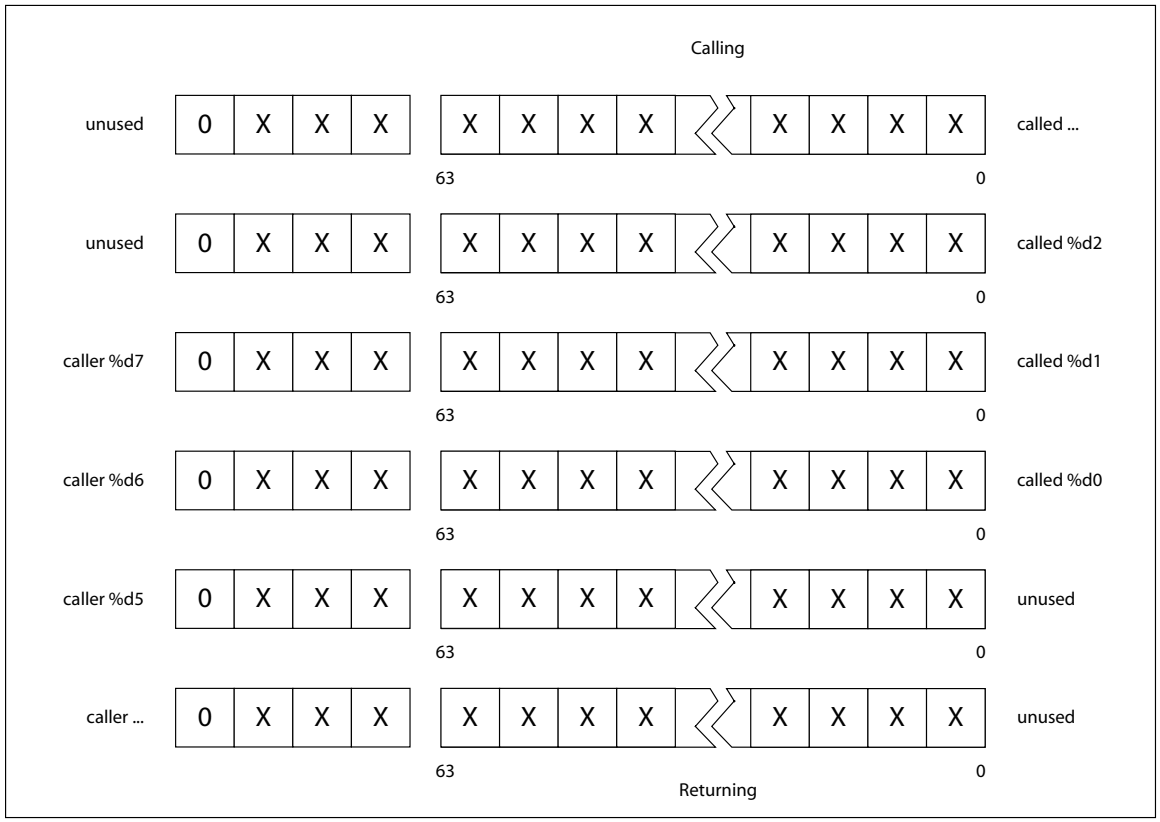

Figure 6.5: A ++VM register window

values. If a free register is not available when a method is called, a register is freed by writing its contents off to memory. Additionally, register windows also allow ++VM methods to share registers. If a method only uses only three data registers, there is no need to save the unused registers; instead, these registers can be given to the called method. As can be seen in figure 6.5, the caller method's data registers %d6 and %d7 are not used and are made into the called method's data registers %d0 and %d1 respectively.

6.2.2 Rotation

When a register window is rotated, it conserves to registers and rotates out six. The lower six registers, registers %d0 to %d5 in figure 6.5 are shifted out, the upper two registers are renamed to be registers %d0 and %d1 in the new frame, and six new

registers be come registers %d2 to %d7. Reference register windows function in the same way as data register windows. Since information such as the stack pointer and frame pointer are stored in the machine registers, the registers %r6 and %r7 do not have special significance in the ++VM. For this reason, rotating out %r6 and %r7 will not cause the virtual machine to reach an inconsistent state.

Rotation Independence Data registers and reference registers rotate independently. Rotating the active data registers does not require rotating the active reference ones, and rotating the active reference registers does not require rotating the active data ones. When possible for a method call to rotate only the data register window, only the reference register window, both, or neither.

Use Register windows are used when calling and returning from methods. When a new method is invoked, its parameters can be stored in registers %d6, %d7, %r6, and %r7 in the caller window. When the method is invoked, these registers are renamed to be the lower-numbered registers of the new method and other registers are made to be the higher-numbered registers. Similarly, when a value is returned, it must be placed in register %d0, %d1, %r0, or %r1 of the current register frame, and, when the window is shifted back the appropriate amount, are renamed to be the higher-number registers in the caller frame. For a more detailed description about how register windows are used when calling methods, see Section 9.3.1.

6.3 Features of Registers

The register nature of the ++VM has many benefits.

Hardware Similarity Today, many hardware architectures are register-based, not stack-based, and can take advantage register-based computation. A virtual machine that is register based will run more efficiently on this form of hardware, just-in-time compilers will not be needed to convert from a stack-based architecture to a register based one. The implementing virtual machine will not have to spend the same amount of effort on register assignment when the just-in-time compiler is run as it would if it had to assign registers from scratch.

Register Assignment Hints When a stack-based virtual machine is run on register-based hardware, register assignment must be done from scratch. Most virtual ma-

chines assign registers at runtime, using a just-in-time compiler using runtime information when deciding what values to put in which registers. The execution of this algorithm slows down code execution, which is needlessly inefficient. Static compilers are capable of assigning values to registers, and they may be able to do a better job than the runtime system because they can work with the semantics of the original code.

Field Assignment Hints A more expressive specification would allow for objects in which certain fields are assigned to registers. If an object is manipulated frequently, it may place some of its values in registers at compile time. For this reason, it may be useful to annotate particular fields in an object, leaving the assignment annotations directly in those particular fields or making them part of a method which accesses that field. While a just-in-time compiler may be able to improve upon this statically-assigned allocation scheme with information gathered at runtime, a statically assigned register scheme may be a more effective place to start than a completely blank slate.

Backup Stacks If a method uses more data or reference variables than it has data or reference registers, the ++VM has a stack to store intermediate values. While a stack is available, it is not intended to be the primary repository for intermediate values. Most operations are optimized to use values in registers, as register operations will be faster than stack operations on most machines and may require fewer opcode words in the instruction set. The logical stack grows and shrinks in alignment with the register window, and provides a backup storage area for the window.

Register Windows Register windows allow registers from one method to be passed directly to another method, requiring only a change of name and not a copying of data. Register windows should improve performance, since hosts will not have to waste cycles in moving data. On hosts that support register windows, this can be implemented natively. On hosts that do not support register windows, those registers that are saved in the virtual register window do not have to move data around whenever a new method is called as long as the host machine can efficiently reorganize register values. The advantage of using a system of register windows is that it minimizes the amount of data that must be passed around internally. By simply renaming registers, it is possible to pass parameters to method with a minimal amount of data movement. A physical machine must save the contents of the oldest register to a backup stack when a new register in a system of register window is needed. The ++VM virtual machine, on the other hand, lets the hardware implement the actual

frame depending upon its number of registers.

6.4 Memory

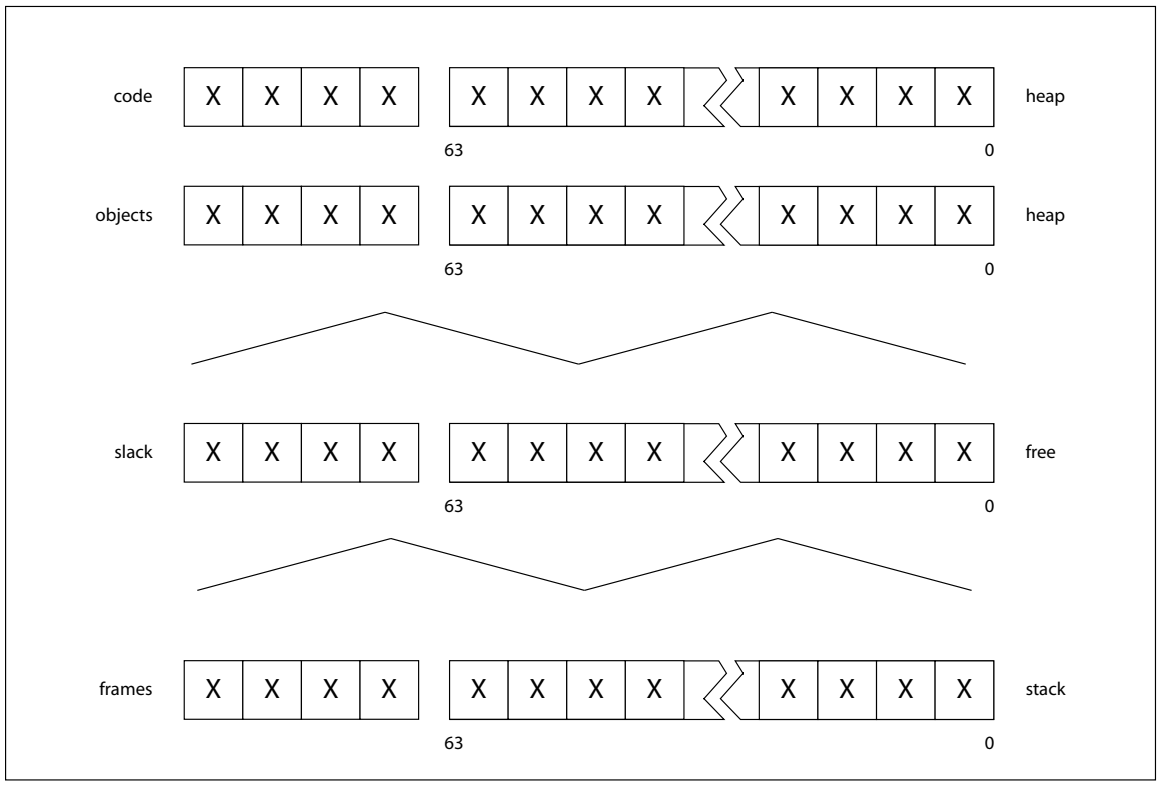

Figure 6.6: The ++VM memory layout

All references in the ++VM are 64 bits. This allows ++VM programs to access multiple exabytes worth of memory, a quantity much greater than past virtual machines have allowed and which will provide ample support for programs in the near future. Though there are certain restrictions on how memory should be used - for instance, all objects must be aligned on long word boundaries - language designers have few restrictions as to how they organize the memory space.

Code The code needed to execute methods in the ++VM are stored in code object arrays. Code objects, like all other objects in the ++VM, have an object header, an

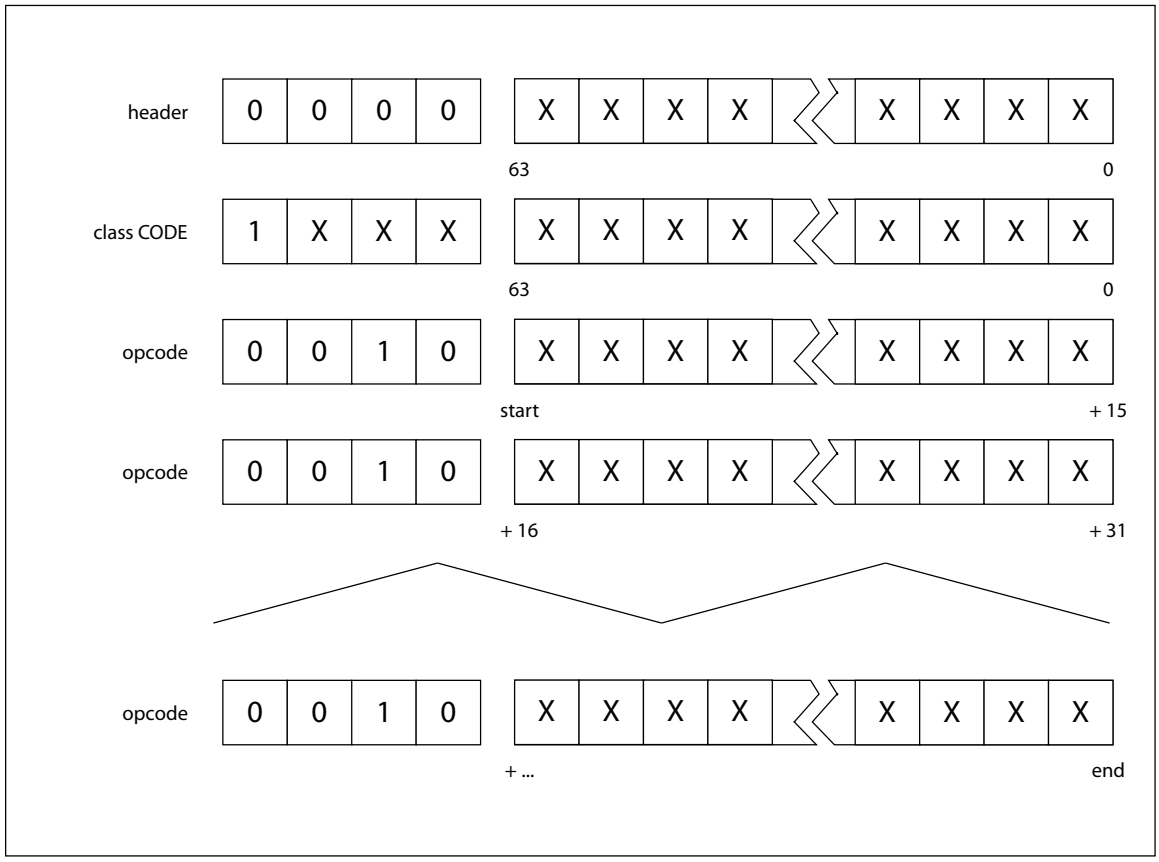

Figure 6.7: A ++VM code object

associated object pointer, and are movable. Because code is stored using the object format. The format of a code object is shown in figure 6.7. For more information about how the code object is used during code execution, see Chapter 9.

Heap The heap is the memory location where most new objects are allocated, stored, used, and destroyed. Strictly speaking, the code region is a subset of the heap. For more information about how objects use the heap, see Chapter 5.

Stack The stack contains information about a thread's behavior. It consists of a sequence of call frames, each containing information about a single method. For information about how methods use the stack, see Chapter 9.

Pinning In the ++VM, objects can be pinned in order to allow pointers to safely traverse the object, or to facilitate interaction with external methods. Objects can be pinned in place, can be pinned to a location at the virtual machine's digression, or can be pinned at a pre-specified location. While many virtual machine implementations may find it more efficient to pin particular objects, such as code objects, the ++VM machine specification does not require that they be pinned.

6.5 Intermediate Value Movement

Registers cannot hold all intermediate values needed by the virtual machine. For this reason, the ++VM virtual machine provides a number of ways to transfer information between registers and memory. There are three important transfer mechanisms: transfers between registers, object-oriented memory transfers, and pointer-addressed memory transfers.

6.5.1 Inter-Register Transfers

The ++VM has a single opcode for moving data and references between registers. This opcode is more compact than the opcode needed to move information from registers to main memory and back. Using this opcode, it is possible to transfer data between registers as long as the tag of the value remains the same. To change a value from a reference to a data type, it is necessary to use the convert instruction. To change a value from a data type to a reference type, it is necessary to perform a cast operation.

6.5.2 Object-Oriented Transfers

The object-oriented design of the ++VM facilitates information access via an object offset. When accessing a field via an object offset, the location of a data value is computed using an offset from the object's header.

Using object referencing in the ++VM requires three parameters: the object, the offset index, and the register which is being loaded or stored. The object being dereferenced must be in one of the reference registers in the current frame or one of the supporting thread registers. Certain operations may require an additional extension word to specify the particular object being dereferenced. Additionally, the virtual machine must know the offset index within the specified object. If this offset is stored in a data register, then no extension word will be needed. If the index is

small, however, an extension word can be used to hold the object offset. Lastly, the source or destination register must be specified; this is the register from which data will be retrieved or to which data will be stored. The virtual machine will know whether or not to use a data register or a reference register based upon the tag bits of the location in main memory.

6.5.3 Pointer Addressing

Unlike other virtual machines, the ++VM allows pointers and pointer arithmetic. This makes it possible for the opcode stream to contain pointers directly to main memory as opposed to indirectly via an object-oriented access mechanism. Certain languages may not use pointers for type safety reasons, but the feature is provided to allow more flexibility and expressiveness to compilers that target the virtual machine. Using pointer addressing requires two parameters: the pointer and the register that is being loaded or stored.

The pointer addressing instructions take their operands from one of four places. Most often, it will a pointer directly from a value already loaded in a reference register and compute its memory value directly. In this case the opcode can be contained in a single opcode word. It is also possible to use pointers as indices, however. Pointer lookup can also be performed using constant a constant offset from the instruction stream or the constant pool; these offsets are stored in one 32-bit extension opcode. To perform more complex forms of pointer arithmetic, it is necessary to move the pointer to a data register, manipulate it, and recast it as a pointer in a reference register.

When using pointer addressing, it is not necessary to specify whether the memory location should be moved to a data or an address register. The virtual machine will use the memory tag bits to determine whether to use a data register or reference register. If the target field is a data field, it will be moved into the appropriate data register. If the target field is an reference field, it will be moved to the appropriate reference register.

Since dynamically allocated objects can be moved around by the garbage collector, a method must ensure that an object has been pinned before using pointer arithmetic. It can do this in one of two ways. First, it can require that the object be pinned on creation by setting a flag in the creation instruction. This will ensure that the garbage collector does not move the object during its operation, but may be too restrictive as an object may not always be subjected to pointer arithmetic during its entire lifetime. Alternatively, a method can request that an already-created object be pinned in place. This will allow a method to use an object as if it were pinned on creation, but allow

the object to be unpinned and moved about at a later time.

6.5.4 Memory-to-Memory Transfers

In the ++VM, it is not possible to move information directly from one memory location to another without using a register. It is first necessary to load the information to a register using a load instruction, then store the information using an additional instruction. The just-in-time compiler may be able to optimize this sequence of operations when the virtual machine is running.

6.6 Features of Memory

Tagged Memory All memory locations in main memory are tagged; this simplifies some memory movement instructions. For more information, see Chapter 7.

Polymorphic Instructions The ++VM has very few instructions that deal with main memory, and implementation of the memory movement instructions should be straightforward. The memory movement instructions are polymorphic, meaning that the type of the data being moved is not specified; the virtual machine is able to infer the data type using the tag bits of the value's operands. During just-in-time compilation, the virtual machine can use the tag bits to decide whether data being moved is a byte or a long and generate the appropriate sequence of hardware instructions.

Pointer Operations In contrast to many virtual machines today, the ++VM allows pointer operations. Like C# and other recent virtual machines, the ++VM allows pointer arithmetic and arbitrary pointer casts. While some languages may choose to disallow pointer operations, especially languages that wish to guarantee type safety, the ++VM machine provides the functionality to those virtual machines that wish to have this flexibility. As a corollary, the ++VM also provides support for absolute addressing, allowing a program to access any arbitrary location in the memory space. Such operations, of course, should be considered with caution.

Chapter 7

Tagged Memory

Tagging memory is a useful way of differentiating between data and references in memory. To a virtual machine, an untyped collection of data bits may be indistinguishable from a pointer; in order to ensure that pointers and data are not confused, it is necessary to include additional bits to type memory. Typed memory, for instance, significantly helps the work of a garbage collector, as a garbage collector most follow object pointers but ignore data words that look like pointers.

There are several differences between the tagged memory system in the ++VM and those found in other high-level language virtual machines. First, the tags in the ++VM are much more expressive than those found in other languages. Though this imposes a larger memory overhead, it creates a much more powerful and flexible tagging system. Additionally, both data and reference types are tagged, a property which allows the virtual machine to be more aware of what it is storing in memory. This facilitates the operation of polymorphic operators and other virtual machine features.

The remainder of this chapter specifies the implementation of tagged memory in the ++VM, describes how memory tags are used, and concludes with a discussion of the benefits of tagged memory.

7.1 Tag Implementations

In the +++VM, every 64-bit long word is augmented by a four-bit tag. This tag determines the type of the long word. All locations in main memory, as well as all registers and all stack values, must be tagged. Tag values for memory that has not been allocated, or for memory values that have been freed by the garbage collector,

are undefined and may be implementation-dependent. An example of the logical tag implementation is shown in figure 7.1.

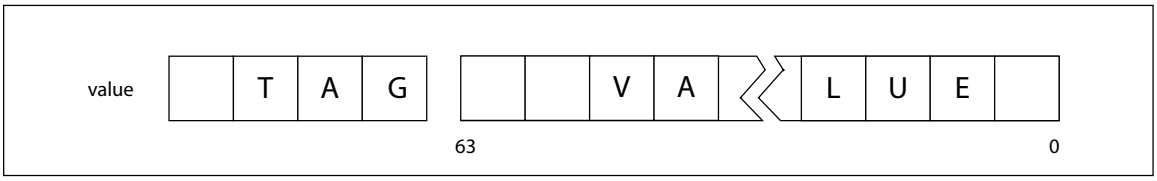

Figure 7.1: A ++VM tag

Data Tags If the most significant tag bit is a zero, then the long word contains data values. Certain settings of the tag bits will indicate that the 64 memory bits are either eight bytes, four ints, two words, one long, or half of an ultra, two floats, or one double; one tag is reserved for future data extensions. For more information about the specific implementation of data tags, see Appendix A.

Reference tags If the most significant tag bit is a one, then the long word contains an object reference or a pointer. Certain settings of the tag bits will indicate that the 64 memory bits are a reference to an object, a reference to an object with code, a reference to a class, a reference to a list element, or a generic pointer; one tag is reserved for future reference extensions. For more information about the specific implementation of reference tags, see Appendix A.

7.2 Tag Usage

Tags are an essential part of ++VM objects and are very influential in memory allocation, memory use, and memory reclamation.

7.2.1 Setting Tags

The manner in which a tag is set depends upon whether the tag specifies a value in main memory or an intermediate value. When a heap object is allocated through the use of the creation instruction, the fields in the objects are tagged according to the static typing of the class. Tags for intermediate values, in registers or on the stack, are much more flexible. These tags are set whenever an appropriate value is moved into that location and can be modified by multiple instructions.

7.2.2 Using Tags

Loading Tags are used to determine the source or the destination of a movement instruction when moving data from main memory to an intermediate value. The virtual machine chooses between using a data or a reference register for its load and store instructions based upon the type of the main memory location. If the main memory location were tagged as a data type, a data register is chosen and the register is tagged with the same value as was the memory location. Likewise, if the main memory location were tagged as a reference type, a reference register is used and the register is tagged appropriately.

Storing Tags can also be used to determine the source of a store instruction, if the store instruction uses an object offset. Because all locations are typed according to the object layout, the virtual machine will be able to determine the tag value of the store destination and choose the appropriate register type to store there. If the memory location is tagged for data, the virtual machine will store the appropriate value of a data register at that location. Likewise, if the memory location is tagged for references, the virtual machine will store a reference value there.

Casting Register tags can be changed by using a variety of conversions and casts. A convert operation changes an argument in a data register to another data type. A cast operation changes a value in a data or reference register into a reference. In both cases, the new tag of the object is specified in the instruction. If the changed value is stored in main memory, it is necessary to ensure that the value is placed in a location with an appropriate tag; it is not possible to directly cast values in main memory.

Selecting Tag bits are used to select the arguments for a polymorphic opcode. As the ++VM does not store the operand types in the opcode stream, it must infer them from the tags of the arguments. For instance, the polymorphic ADD opcode determines whether to bytes, shorts, words, longs, ultras, floats, or doubles based upon the tags of its operands. Furthermore, tag bits are used to differentiate between objects with different properties. As objects may have differing associated object pointers and may not have code pointers, the memory tags can be used to determine how an opcode should behave. For instance, the method-invoking opcode CLL will behave differently if an object has a code pointer; if the tag bits show that an object has a code pointer, then the optimized code will be called rather than a class's generic code. More information about individual opcodes, and how they use the tag bits, can be found in Chapter 8.

Trapping Tag bits are incredibly helpful in auto-boxing and class trapping; in the ++VM, it is possible to invoke methods on primitive data types. When this method is invoked, the virtual machine must check the field's tag to ensure that the method is being invoked on an object value. If the method is not invoked on an object, then the virtual machine traps to an appropriate location, calling the appropriate method on the primitive object. Memory tagging thus automatically boxes primitive data types, associating them with a particular class without requiring a wrapping object. For more information about auto-boxing and class trapping, see Section 9.3.2.

7.2.3 Releasing Tags

The process for releasing a tag depends upon whether the tag is associated with a value in main memory or with an intermediate value. For values in main memory, the tag is immutable until the object with which it is associated is released or moved; the tag reclamation process is automatically performed by the garbage collector. Intermediate values, on the other hand, can change from instruction to instruction and do not have to be explicitly set.

7.3 Tag Features

Tagged memory is very powerful. For a 6% overhead, the ++VM is able to provide polymorphic operators, short instructions, data type distinctions, reference type distinctions, packing, verification, and garbage collection support. As much of this overhead is already present in many systems with small, dynamically collected objects, the costs of providing tagged memory should be minimal.

Polymorphic Operators In the ++VM system, opcodes do not operate on a specific data type. Instead, an opcode defines a kind of abstract operation, and the memory locations will determine the actual code that is executed. This polymorphism allows the opcodes to be as general as possible since the virtual machine will be able to infer type information from the tag bits. Additionally, using tagged memory values creates a more compact instruction set and frees additional bits to for instruction attributes.

Short Instructions By using a tagged memory system, the ++VM frees a number of bits in the instruction word. Because the instructions in the virtual machine do not have to specify the data types being manipulated, instruction bits are freed in

order to hold attributes. As these attributes can improve code execution speed, the memory tagging can lead to overall performance increases.

Data Distinctions Tagged memory allows the virtual machine to distinguish between different types of data. Memory tagging helps the virtual machine distinguish between the data elements even in the absence of static type information. Most other virtual machines do not distinguish between different data types at the memory level, instead relying on the instruction set to distinguish between data types. The ++VM, on the other hand, explicitly specifies the primitive data type for every data element in the system, a fact that may enhance runtime security and debugging where it is important.

Reference Distinctions Additionally, tagged memory helps the virtual machine determine the size of an object. Reference types can be of different sizes, varying from pointers to full-blown objects with locks. Using tagged memory, the virtual machine can determine various properties of a reference type and adjust its behavior accordingly. For instance, the tag bits can help a method invocation opcode select an optimized method for an object, if one is available, rather than the general method available through the object's class. By keeping track of the size and components of an object, the memory tags allow for objects that can take better advantage of the opportunities available in the system.

Packing Because the ++VM tags every 64 bit region, it is not necessary to tag every byte, every short, or every int. This allows the virtual machine to pack multiple bytes, shorts, or ints into a region which use a common tag. While it is the compiler's job to perform this packing optimally, the virtual machine benefits because it can combine tag bits for multiple data values.

Enhanced Verification Current languages ensure type safety by using verifier in order to ensure that objects and data are treated appropriately. Verifiers use static language properties in order to ensure that operations are performed on values of the appropriate type, making sure that all operations in the virtual machine do not violate the properties of the language. Using tagged memory augments the power of a static verifier, allowing it to determine certain virtual machine properties at runtime. Certain programs may contain legitimate runtime behavior but can rejected by the verifier because they violate the static constraints of a language. By using a tagged memory system to augment the static verifier, the virtual machine may be able to

increase the power of the verifier. It should be noted that some systems, including the Java virtual machine, must implement typing to perform static access verification.

Garbage Collection In a garbage collector, the virtual machine must determine whether a value is a reference. By using a tag bit to determine whether a memory location stores a reference type or a data type, the garbage collector can determine whether a sequence of bits must be searched for further references. In the absence of tagged memory, garbage collectors must be extremely cautious: often, conservative garbage collectors will mistake data for objects and will fail to free certain memory locations. By using tagged memory, garbage collectors can avoid this problem and collect all relevant garbage.

Eventual Hardware Support Tagged memory is becoming increasingly important for current systems, not just the ++VM. Most garbage-collected software systems, not just the ++VM, require some form of memory tagging in order to differentiate between objects and pointers. Most hardware, however, does not currently support tagged memory. By explicitly specifying the need for typed memory in the virtual machine specification, it is hoped that hardware designers will begin to realize that tag bits are important to advanced memory systems.

Chapter 8

Opcodes

Loosely defined, an opcode is a basic unit of work in a virtual machine. Complicated programs in a language are compiled down to a sequence of opcodes to be executed, as the opcodes serve as the machine language of a virtual machine. In current systems, the opcodes are generally machine-independent, meaning that they are not meant to be directly executed in hardware. While some opcodes can be directly mapped to hardware operations on some machines - an example is the addition instruction, which can be found in almost all real and virtual machines - more complicated opcodes, such as those for calling methods or switching threads, are generally handled by software as opposed to hardware.

Opcodes for the ++VM are meant to be more general and more versatile than opcodes for other high-level language virtual machines. First, opcodes in the ++VM are generally more expressive than those for other languages, providing optional information about the context of the operation being performed. Though this imposes a small size overhead, it allows for potential increases in code execution speed. Additionally, the ++VM provides a number of operations that are not found in other virtual machines. These operations, such as set operations and synchronization points, give the compiler access to a number of powerful tools.

The remainder of this chapter specifies the format of opcodes in the virtual machine, then describes the process by which opcodes are executed. It then provides an overview of the types of opcodes that the virtual machine can execute, and concludes with a discussion of the features of the ++VM instruction set. Full details of the instructions can be found in Appendix B

8.1 Opcode Format

As shown in figure 8.1, each opcode is a sixteen bit word. This word can be divided into two sections: an eight-bit instruction byte and an eight-bit attribute byte. Some opcodes may require additional words which hold additional attributes or additional opcode information.

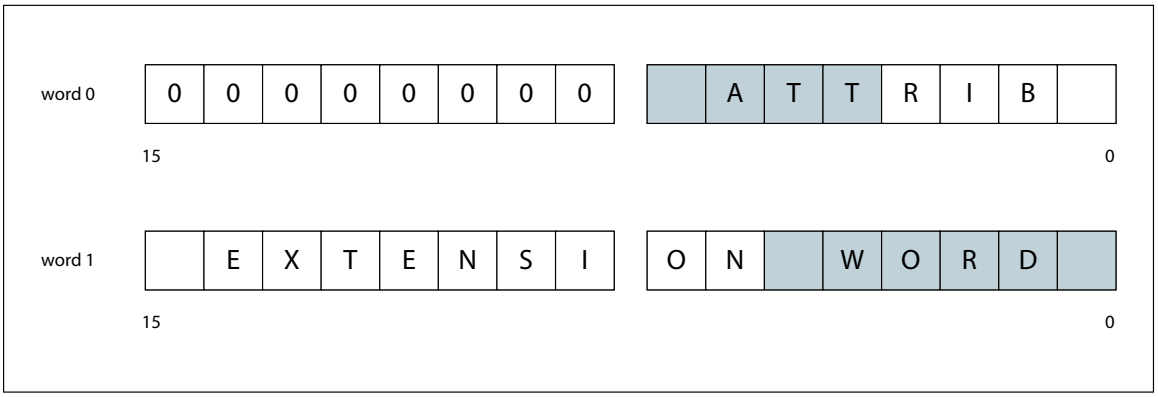

Figure 8.1: A ++VM opcode

The first eight bits of the opcode, the zero bits in figure 8.1, specify the instruction byte that the virtual machine will execute. If the instruction byte is less than 64 (if the bytecode begins with two zeros), then the byte specifies one of the sixty-four "intrinsic" instructions of the ++VM, the instructions that are defined by this virtual machine specification. These instructions are central to the functioning of the virtual machine and are defined later in this chapter. If the instruction byte is 64 or greater, the opcode is an "extrinsic" instruction and is a reference to a code group. These code groups are defined at the method, frame, or thread level and will change between classes. More information about extrinsic instructions and code groups is available in Section 11.1.

Regardless of whether the opcode is intrinsic or extrinsic, the next eight bits of the opcode are the attribute byte, as can be seen in figure 8.1. This region of the opcode that holds more specific information about the instruction. Depending upon the specific opcode, bits in the attribute byte may specify source and destination registers for an operation, may hold short constants, or may hold instruction-specific attribute information. Many of these bits are required, such as the bits a with white background in figure 8.1, but other bits are optional information attributes and are depicted against a gray background, a depiction that is uniform throughout this

thesis. The information encoded in these attribute bits is highly dependent upon the type of opcode. More information about these attributes is available in Chapter 10.

Some opcodes may have one or more additional extension words, such as word 1 in figure 8.1. Like the attribute field in the original opcode instruction, these bits help give more information to the virtual machine. The information in these bits also varies depending upon the type of the instruction. For instance, these bits can contain indices into the constant pool, pointers to relevant registers, or other optional attribute information.

Opcodes specified in the ++VM should be stored and transmitted in network byte order for platform independence. Internally, however, implementers are free to treat ++VM instructions in an appropriate manner.

8.2 Opcode Execution

Opcode execution or just-in-time compilation is a six step process: fetch, classify, extend, type, execute, and write back.

Fetch When fetching, the virtual machine must decode the instruction. This is done by fetching the first word in the opcode stream and examining the instruction byte. The instruction byte specifies the type of operation that should be performed by the virtual machine.

Classify In the typing stage, the virtual machine must determine the opcode family in order to execute the proper instruction. This is done by parsing the instruction byte of the opcode. If the opcode value begins with two zeros, then the opcode is one of the 64 basic opcodes of the ++VM and should be parsed appropriately. Otherwise, the opcode is reference to a code block. If the operand is one of the basic opcodes, the virtual machine executes the appropriate code; otherwise, the virtual machine determines the location of the code block and prepares to execute the bytecodes stored in that location.

Extend Once the opcode is understood, it is necessary to retrieve any additional instruction words that contain instruction attributes. From the information available in the first instruction word, the virtual machine can determine whether it is necessary to fetch additional words from the instruction stream. As some opcodes consume multiple words, the virtual machine must retrieve all attribute information about the opcode before beginning to execute it.

Type Before the instruction can be executed, it is necessary to examine the tags of the instruction input values. For some instructions, such as the add instruction, this is necessary to determine the specific instance of the polymorphic instruction that must be generated; the tag specifies the size of the data values. For other instructions, such as the call method operation, this is necessary to determine whether the address value is a future that still needs to be computed.

Execute Once the virtual machine understands what type of instruction to execute and has checked the appropriate tags, the instruction can be executed. Any information contained in the instruction's informational attribute bits must be used when executing the instruction; information contained in the informational attributes can be ignored if the ++VM implementation does not support them.

Writeback Lastly, the virtual machine must update the condition code register with the results of the instruction execution. Some implementations may determine that the next instruction will not look at the condition code register; these machines can skip this step. Once the virtual machine has updated the condition code register, interpretation of the instruction is complete, and the machine can begin the process over again with the next instruction.

8.3 Bytecode Instructions

The ++VM specifies no more than sixty-four opcodes, the intrinsic opcodes of the virtual machine. These can be divided into several broad general categories described below.

8.3.1 Math Operations

The ++VM supports six arithmetic opcodes: addition, subtraction, multiplication, division, modulus, and inner product. These operations work on all data types stored in data registers or on the stack. The additional information fields of these instructions will specify the location of the data operands.

Shift Operations The ++VM has three bit shift opcodes: logical shift left, logical shift right, and arithmetic shift right. These operations work on all data types stored in data registers or on the stack. The additional information fields of these instructions will specify the location of the data operands.

Set Operations The ++VM has five set operation opcodes: set union, set intersection, set exclusive or, set inner product, and bit toggling operations. These operations work on all data types stored in data registers or on the stack. The additional information fields of these instructions will specify the location of the data operands.

8.3.2 Memory Operations

The ++VM has three kinds of memory management operations. These operations move data between different sections of memory, preserving memory tags as the data is moved.

Register Operations The ++VM has a single opcode that is optimized for moving values around between registers, and moving between the registers and the stack.

Casting Operations The ++VM has two type-changing opcodes: conversions, which are used to type data, and casts, which are used to type objects. The convert operation to change a variable to the specified data type. The cast operation will change a variable to a specific class type.

Object Movement Operations The ++VM has two opcodes for accessing fields from an object offset, one opcode for loading and one for storing. The object referenced can be stored in a thread register or in a user register. The source or destination of this movement must be a data or address register.

General Movement Operations The ++VM has two opcodes for accessing fields from a general memory offset, one opcode for loading and one for storing. The offset amount must be stored in a data register. The source or destination of this movement must be a data or address register.

8.4 Control Operations

The virtual machine has three ways to control execution flow: branching (with optional assistance from comparison operations), method calling, and exception handling.

Comparison Operations The ++VM has three opcodes that explicitly set bits in the condition code register. Though all ++VM instructions set the condition code register bits to the result of an opcode's execution, these operations set them explicitly. These opcodes compare values in reference registers, compare values in data registers, and a check bounds on a number.

Logical Control Operations The ++VM has two logical control opcodes, operations that set the program counter to a new value based on the values in the condition code register. One opcode branches on the values of the negative or the equality bits. However, a supplemental branching opcode jumps based on arithmetic overflow or underflow, on the value of the exception bit, or on the value of the future bit. For more information on logical control, see Section 12.1.

Calling Operations The ++VM uses only one instruction for calling methods and only one for returning from them. The call opcode can be used to invoke methods statically, dynamically, or via a wound call. The return opcode is polymorphic. For more information on methods, see Chapter 9.

Exception Operations The ++VM uses two opcodes to handle exceptions. One instruction is needed to create an exception handler which manages exceptions when they occur. Another instruction is needed to throw an exception, to create an exception object that can be handled later. For more information on exceptions and exception handling, see Section 12.5.

8.4.1 Creation Operations

The virtual machine uses only three instructions for creation. These are the create object, create class, and create method operations.

Creating Objects The ++VM has one opcode for creating objects. This opcode is used to create both arrays and objects; the two are differentiated only by instruction attributes. For more information on objects, see Chapter 5.

Creating Classes The ++VM uses one opcode for creating a class out of an array of words. This allows for dynamic class definition, should languages choose to allow this opcode.

Creating Methods The ++VM has one opcode for creating a method out of an array of words. This allows for dynamic method creation and closures, should language choose to allow this opcode.

8.4.2 Thread Operations

The ++VM provides four opcodes that will be useful to multithreaded programs. These are the monitor enter and monitor exit instructions, the yield instruction, and the synchronization point instruction.

Monitors The ++VM provides two opcodes for thread locking operations. These are the opcodes for entering and exiting a monitor, and a per-object lock that are contained in the object header.

Thread Switching The ++VM provides two opcodes for yielding thread control explicitly. The yield opcode will suspend a thread, possibly signaling another thread in the process. The synchronization point opcode sets up a common checkpoint across multiple threads.

8.4.3 Annotations

The majority of the remaining opcodes are built-in annotations. They provide more attribute information about the running code, helping the virtual machine make more informed decisions about how and where to optimize.

Loops The ++VM has one opcode for explicitly specifying the boundaries of a loop. This opcode also provides hints about the loop runtime behavior. For more information on loops, see Section 12.3.

Code Groups The ++VM uses two opcodes to bind a collection of opcodes or a library call to a single block opcode: one opcode to start the group and specify a number of attributes and another opcode to end that group. In conjunction, the two opcodes will associate an arbitrarily large group of opcodes to another opcode; this new opcode can be scoped at the method, class, or thread level. This new opcode can then be used in the instruction stream to reduce code size and increase performance. For more information on code groups, see Chapter 11.

Object Attributes The ++VM has an instruction for providing additional information attributes for an object. Using this opcode, the compiler can provide additional hints about an object's behavior over its life cycle.

Annotations The ++VM uses one opcode to specify the onset of an annotation. An annotation is an interface that allows programmers and virtual machine designers to define their own additions to the virtual machine. Unrecognized annotations may be ignored. For more information about annotations, see Section 10.3.

8.4.4 Miscellaneous Instructions

Five instructions do not fall into other general categories.

Stack Operations The ++VM has one operation that performs stack modifications, such as pushes, pops, value duplications, and value reversals.

One Operand Instructions The ++VM has an opcode that performs a variety of operations which only need a single operand. These include one's complement negation, two's complement negation, population count (number of set bits), leftmost bit index, rightmost bit index, trap vector assignment, and object size count.

Illegal Instruction The ++VM has an illegal instruction, an instruction that will kill the virtual machine if executed. It exists so that the virtual machine can avoid executing data.

8.5 Features of Opcodes

The ++VM opcode set has many useful properties, including a small instruction set, polymorphic operations, informational attributes, extensible instructions, future support, set operations, architecture independence, and language independence.

Small Instruction Set The base instruction set for the ++VM is very small, especially in comparison to the instruction sets of most virtual machines today. All of the built-in opcodes to the ++VM can be represented using only six bits, meaning that the virtual machine is designed to execute only 64 distinct opcode families. However, the current ++VM specification only defines 44 distinct opcodes. This

number is barely a quarter of the number of opcodes defined in Java and five percent of those needed for large languages languages such as C#. These small instruction sets have a number of desirable properties. First, in addition to general elegance, a small instruction set is RISC-like in that every instruction in the ++VM is considered an essential feature of the virtual machine. Additionally, a small instruction set makes sense from a conceptual standpoint: while computers care whether two addends are shorts or long longs, from a human standpoint there is little inherent difference. From a more practical standpoint, a small pre-defined instruction set leaves a host of instructions available for block and library operations, which allow the virtual machine to be customized by the user.

Polymorphic operations Most instructions in the ++VM are polymorphic, meaning that they can operate on values of varied tags. Thus, while many languages use a large number of opcodes to differentiate between different flavors of a specific instruction, the ++VM uses its tagged memory system to select the appropriate instruction in an opcode family. This use of polymorphic operations is helpful for both data and reference values. Data values can use the memory tag to determine which type of operation to perform; for instance, the polymorphic add operation uses the tags of the operands to generate the appropriate hardware instructions for the addition of bytes in contrast to addition of ints. Reference values, can use the tag in a variety of ways, for instance to differentiate between futures and computed objects or between generic objects and list elements. Because of these polymorphic operations, the virtual machine does not have to have a collection of special opcodes to deal with these features. Additionally, polymorphic operations may be useful in cases where a static compiler may not know the static type of an object, which can be true with futures. Though some polymorphic instructions may require more runtime or compile time in order to ensure code correctness, the benefits far outweigh this slight performance penalty.

Informational Attributes One of the most important features of the ++VM are the informational attributes found on many instructions. These are optional attributes that convey more information about the context of the instruction. The virtual machine implementation can ignore these attributes without causing the code to perform incorrectly; however, by taking advantage of the informational attributes these implementations can improve performance. As of this writing, no other high-level language virtual machine includes this information directly in the opcode stream. For more information about attributes, see Chapter 10.

Extensible Instructions The ++VM instruction set is designed to be extensible in three major ways. First, since only 44 of the 64 opcodes are currently defined, there is plenty of room to expand the opcode set. Additionally, many instructions have an extension bit, which makes it easier to add additional information attributes to an opcode without requiring a rewrite of the instruction set. Most importantly, though, the ++VM allows for code groups. The virtual machine can bind a user-defined or library-defined block of code to opcodes, allowing for extensibility at runtime. This feature is not found in other virtual machine languages. For more information about blocks and libraries, see Section 11.1.

Future Support The ++VM defines a system of opcode-level support for futures or promises. Futures are fields whose value have not yet been computed, and can be extremely useful for parallel and lazy evaluation. Though only a few high-level languages use futures, futures are powerful features that can make potentially make computation more efficient. The ++VM provides a number of opcodes to create, test for, and run futures, providing full support for languages that choose to use futures, something that is not often found in a high-level language instruction set.

Set Operations The ++VM provides opcode-level support for set operations. The majority of current high-level language virtual machines do not provide support for these operations, instead relying upon a series of shifts and logical instructions in order to toggle a single bit in a number. In the ++VM, sets are supported with such features as a population count (which returns the number of ones in a number), features that are easily implemented in current hardware but are not always directly supported by high-level languages.

Architecture Independence The ++VM instruction set is not designed to be implemented on a particular architecture. Though the virtual register nature of the virtual machine favors register-based real machines, the ++VM is flexible enough to be implemented on any architecture. All instructions are generic and hardware-independent.

Language Independence The ++VM instruction set is not designed to implement a particular language; instead, it is designed to support as many high-level languages as possible. Languages implemented on the ++VM will use some subset of the opcodes specified by the virtual machine: some languages will not allow for the execution of certain opcodes entirely, while others will ignore certain attributes of

other instructions. For instance, type-safe languages like Java may not take advantage of the pointer instructions for security reasons.

Chapter 9

Methods

9.1 Methods in the ++VM

A method is the basic grouping of code in many high-level languages. Many high-level languages prefer to use a large number of rather small methods, making the method-calling overhead an important factor in the performance of high-level language virtual machines.

Methods in the ++VM are designed to be more expressive than those in other languages. First, it provides a simple but powerful interface for calling methods. Though it uses only two opcodes to support methods, the ++VM introduces a number of ways to call and return from methods which have not been used in existing virtual machines. Furthermore, the virtual machine provides a powerful set of method attributes. These attributes can improve method performance and increase the overall efficiency of the virtual machine.

The remainder of this chapter describes the process of calling a method and the process of returning from a method. It then provides an overview of the attributes which can optimize method execution and a more detailed description of a new method-calling procedure, the wound call. The chapter concludes with a discussion of the features of methods in the ++VM.

9.2 Kinds of Methods

The ++VM provides opcode-level support for four manners of calling methods: virtual methods, static methods, wound methods, and trapped methods.

Virtual The code for a virtual method is determined by the runtime type of an object. When the virtual method is invoked, the virtual machine determines the code to be executed by examining the objects - if it has JITed code - or the class's virtual method vtable. While a virtual method is running, the this pointer is set to the object on which the virtual method will be invoked; the this class pointer will be set to that object's class.

Static The code for a static method is determined by the compile-time type of an object and do not depend upon class instances at runtime. At compile time, the virtual machine can determine the appropriate code to execute because the code is not dependent upon runtime information; this means that static methods will be more efficient than virtual methods as they will not need to look up their code in a vtable. While a static method is running, the this pointer will be set to null, since static methods cannot be called on object instances; the this class pointer will be set to the class that contains the static method.

Wound The code for a wound-call is determined by the static type of an object; unlike a static method, however, wound calls are performed on an object. For instance, object initialization in Java is an example of a wound call. A wound call is head recursive: when a wound call is invoked, the virtual machine first attempts to call the wound call on all ancestor classes of the current object before calling it on this one. Wound calls attempt to reuse the method frame to improve performance. For more information about wound calls, see Section 9.7.

Trapping This code for a trapped method is determined via a trap vector for an object. Whenever any method is invoked, the virtual machine uses the type bits to verify that the value is an object as well as whether the object has a class pointer or whether it has code. If the value is not an object, the virtual machine will trap it and dispatch a method based upon a class trap vector; this process allows the virtual machine to invoke methods on primitive types and on list elements. When a trapped call method is invoked, it serves as a hint to the virtual machine that the invocation will trap and that the host machine should branch-predict accordingly. While a trapped method is executing, the this pointer is set to null, since the caller value is not a standard object; the call pointer is set to the appropriate class from the trap vector.

Others While languages that run on the ++VM may use other method-calling procedures, the ++VM does not provide bytecode support for them. Other types of method calls, such as those for interface methods, can be created from a combination of ++VM instructions and the above method calls. It is expected that one of the ++VM libraries will provide an optimized routine for these kinds of method calls. Those programmers and languages that wish to support more complicated method invocations should use the calls provided in that library.

9.3 Method Behavior

9.3.1 Calling Methods

Calling a method is a multi-step process, consisting of parameter placement, call parsing, code location, method execution, and method return. The general process is the same under both interpretation and just-in-time compilation execution schemes.

Parameter Placement Though technically not part of the execution of a method, it may be necessary to place parameters in a location for the method to access. Because register windows are used along with methods, it is possible to pass method parameters by leaving up to two data parameters and up to two reference parameters them directly in the frame of the caller method. When the method is called, the data register window and the reference register window are shifted by default, passing up to four parameters directly to the called method. Additional data and reference parameters will have to be passed via the stack. For more information about register windows, see Section 6.2.

Call Parsing Once the parameters have been properly placed, the virtual machine executes a call instruction. This instruction specifies the behavior of the method call and specifies any information attributes that the method might need. The virtual machine records this information and prepares a new call frame, but does not yet attempt to load the code for the method.

Code Location Once the call frame has been prepared, the virtual machine then loads the method code by checking the tag of the value on which the method is being called. If the tag shows that the value is not an object, the virtual machine should trap to the appropriate value in the trap vector. If the value is an object with code, the virtual machine should use the object-specific code for improved performance.

Otherwise, the virtual machine should locate the code in the object's class and execute the appropriate code.

Method Execution Once the virtual machine has created a frame and determined the code location, it should execute the method. Method execution involves executing the opcodes for the new method until a return statement is reached.

Method Return When the new method executes a return statement, the method frame is released, the this object and the this class pointers are restored, and the register window is shifted to restore the registers of the caller method. Any additional parameters should be popped from the stack.

9.3.2 Trap Vector Operation

The trap vector is used when methods are invoked on non-object values.

Setting Trap Vector Values In order to execute methods on data objects and on list elements, it is necessary to set the class that corresponds to the list element in the trap vector. This is done using a specific one-operand instruction, which equates the list element reference type with a specified type of class. In order to set the trap vector value, a pointer to the desired class is placed in a virtual machine reference register; the instruction then inserts that value into the appropriate location on the trap vector. From that point on, all methods called on the list element data type will correspond to calls to methods of that class, as these calls will go through the trap vector. The trap vector reference is held in a machine register, %tr.

Trapping The virtual machine checks to see whether a value is an object when it checks the tag bits to find a code pointer. If the value is not an object, then the method call will trap to the virtual machine's trap vector. Using the trap vector, the virtual machine will determine the value's class as well as find the proper method to call on that value.

9.4 Method Attributes

The method calling opcode uses a number of attributes that support optimization. These attributes provide a number of hints to the just-in-time compiler and allow the

JIT to make informed decisions about what to optimize. For more information about attributes in general, see Chapter 10.

Inline The inline attribute determines whether or not a method should be inlined. Inlining a method is an optimization in which a method's code is spliced into the method of the caller class and saves the virtual machine the effort of allocating a new call frame for the method. This attribute uses two bits to determine whether the virtual machine should sometimes, always, or never try and inline the method.

Conserve The conserve register determines whether the parameter registers will be conserved after the method call. If the registers are conserved, then they will be addressable by the caller method and may contain additional return values. If the registers are not conserved, then the virtual machine cannot access those registers until a new value is explicitly stored there.

Tail Call The tail call attribute determines whether or not a method should be tail-called. A method that is tail-called will use the same call frame as the caller method because the caller method has finished execution. Providing support for tail calling allows the ++VM to support some forms of recursion more efficiently.

Code Pointer The code pointer attribute is set if the object on with the method will be called is likely to have a code pointer. By setting this attribute, the virtual machine will predict that the check for an object's code pointer will succeed and that object-specific code can be used. Also, setting this bit may be a way to encourage the virtual machine to attach code pointers to objects on which this method will be called.

Common The common attribute is set if this method will commonly be called on objects of the same class as the current object. By setting this attribute, the virtual machine is encouraged to optimize a virtual method for this type of class, either by branch prediction or by inlining. Both of these changes will help speed up code execution.

9.5 Returning From Methods

The return opcode specifies that a method has completed execution and that the caller method should resume execution.

9.6 Return Behavior

The return opcode returns from a method and destroys the current method's frame. This same return statement is used for all types of returns, including methods that return void, methods that return a single value, and methods that return multiple values. The opcode does this by specifying the number of registers to preserve when returning to the caller frame.

Execution To return a void, the return opcode should conserve zero registers. To return a single register, the return opcode should conserve only one register and the register type bit should be set appropriately. To return more than one register, the return opcode should conserve the appropriate number of registers; however, conserved registers must either be all data registers or all reference registers.

9.6.1 Return Attributes

The return opcode has one attribute.

Frequency The frequency attribute specifies how often a particular return will be called. There are three levels of frequency, and these can be used in order to optimize the manner in which methods will terminate in order to help the just-in-time compiler predict likely code execution paths through a method.

9.7 Wound Calls

Description The wound call, outlined in figure 9.1, is designed to provide a way of calling related methods while reusing the same frame. For instance, when objects are initialized in object-oriented languages, an entire call stack is created for the object initialization method, the superclass initialization, and all the other classes up the ancestor list. A wound call is designed to invoke initialization methods - as well as other methods which have the same names, return values, and parameter types - without having to store the entire call stack in memory.

Motivation The primary motivation for the wound call is the object constructor. The ++VM provides the wound call in order to facilitate object creation, a very common operation in object-oriented languages. However, programmers and compiler writers may be able to find a number of novel ways to use this feature. First,

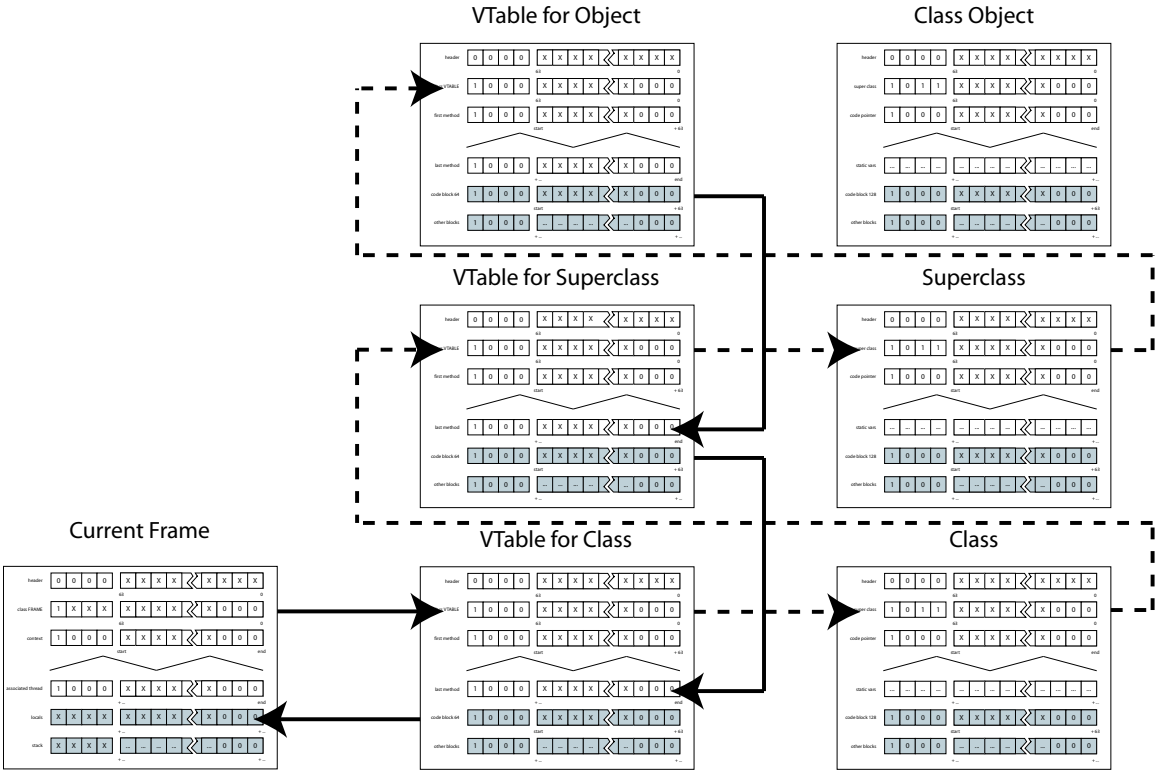

Figure 9.1: Use of a Wound Call

the wound call is not tied to any particular method. This means that the wound call syntax provides a way of "head calling" other essential methods, such as destructors, should programmers wish to do so. Additionally, wound calls will only walk the ancestor list as long as a given class has an appropriate name, signature, and parameter list. This means that wound calls can be used to call methods in user-defined classes only.

Use Figure 9.1 shows the progression of a wound call invocation. The dotted arrows show the progression before any opcodes are executed. First, the virtual machine determines whether the super class contains an appropriate method; in this case, it does. Next, the virtual machine determines whether the superclass's superclass, the Object class, has an appropriate method; in this case, it does as well. Since Object does not have a superclass, the winding is complete. From this point, the virtual machine continues code execution along the solid lines, executing first the code for

Object; when the method in the Object class encounters a return statement, the frame is not recycled upon execution completion. Instead, the frame is re-used to invoke the appropriate method in the superclass, and, when that has completed, for the method of the original class. When the class's method finishes execution, the wound call has completed, the frame is released, and execution continues with the next opcode.

9.8 Features of Methods

JIT Hints Attributes such as the inline attribute or the common attribute can be used to provide hints about code execution. These hints can be used directly by the just-in-time compiler to increase program execution speed. These attributes can further increase execution speed by potentially cutting down on the amount of code monitoring necessary when the program is executing.

Per-Object Optimization The code pointer attribute provides a way to indicate that a method should be optimized for a specific instance of the class. Per-object optimized code can be much faster than generic object code, especially because the virtual machine can make assumptions about per-object runtime constants.

Optimized Calls The ++VM provides an optional per-object code pointer that can contain optimized methods for a particular instance of a class, as described in Chapter 5. When the ++VM calls a method on an object, it should check to see whether that object has an optimized method through its code pointer. If the code pointer attribute is set, the method invocation opcode can prepare the virtual machine so that this check will succeed and execution can proceed much more quickly.

Virtual Method Optimization The common attribute allows a method invocation to indicate that it will often be called on the same type of object. If the JIT compiler can optimize the code assuming that an object is of one particular type, then it can potentially speed up code execution in the future. The common attribute will allow the virtual machine to make assumptions about future method behavior and lead to improved overall performance.

Recursion Support In contrast to virtual machine languages such as Java, the ++VM provides efficient recursion support. Using the tail-calling attribute, the

method invocation instruction can call a method within the current frame. This process of tail-calling methods will save space and time, as the virtual machine will be able to eliminate additional call frames and will not have to copy data needlessly.

Wound Calls The wound call, as described in Section 9.7, allows a virtual machine to make use of the same call frame to call multiple related methods. A wound call first calls a method on an object's ancestor class, then on all parent classes, then finally on the class itself; at each level, the wound call has already been performed on all possible ancestor levels of that class. Currently, high-level languages do not provide opcode level support for wound calls, even though they perform operations with similar functionality; for instance, wound calls can be used to initialize or finalize objects in many languages. By using a wound call, the virtual machine can reuse the frame, saving stack space and copying time. Additionally, the wound call syntax is flexible enough to allow programmers to use wound calls for other, more complicated operations.

Trapping The ++VM uses a trap mechanism to dispatch methods on primitive data types and on list elements. In contrast to other virtual machines, which require complicated wrapper structures for structures without code pointers, the ++VM is able to use the memory tag in order to determine which method to call. Using the tags, it is possible to eliminate the extra space and indirection needed by these wrapper classes. Additionally, the trapping mechanism is flexible enough to be manipulated as the virtual machine is running. This makes it possible to change the type of list element being manipulated, allowing for maximum flexibility, increased space efficiency, and only a minimal performance penalty.

Chapter 10

Code Support

In many computer languages, an offline compiler may be able to determine several characteristics of a program's runtime behavior. However, due to the restrictive nature of a machine's instruction set, this predictive information cannot be given directly to the runtime system and must be inferred as the code executes. If this information can be passed to the virtual machine using optional bits in the opcode stream, though, performance can be improved and just-in-time compilation can be streamlined. While virtual machines can choose to ignore this information, as the suggestions are not vital to correct code execution, they may be able to take advantage of it to increase execution speed.

This chapter focuses on the the ++VM instruction set's two pathways for passing information to the virtual machine. The first way is the use of attributes, optional information bits already associated with opcodes that can provide insight into probable program behavior. The first part of this chapter is devoted to the description of the attribute system and the advantages that various attributes can provide. The other pathway is the use of annotations, more complicated structures that provide increased description of code behavior which take advantage of third-party extensions to the ++VM instruction set. The remainder of this chapter discusses the annotation system and the language features possible using annotations.

10.1 Opcodes with Attributes

In the ++VM virtual machine, opcodes use parts of the the second byte of the instruction word, as well as parts of additional extension words, to store attributes. In figure 10.1, twenty-four bits are available for attributes: these are all of the opcode bits

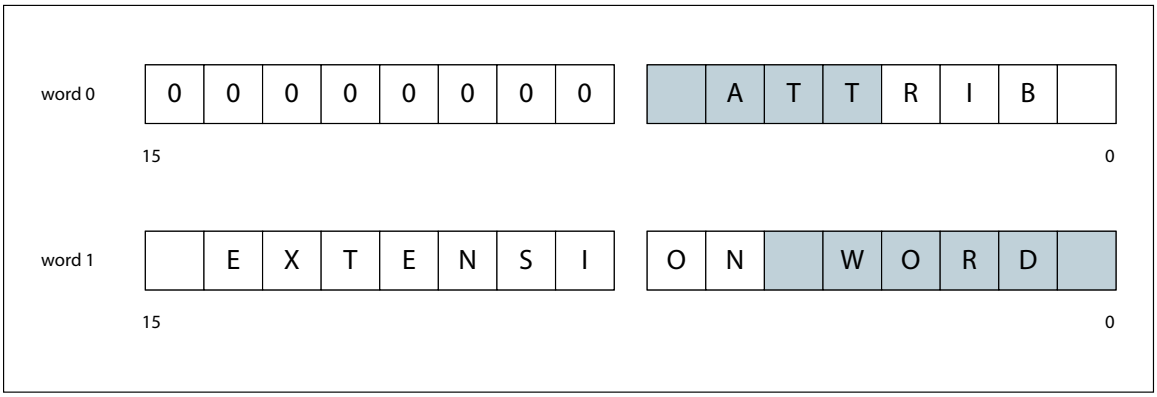

Figure 10.1: A ++VM Opcode

except the eight zeros which specify the opcode type. As usual, required attributes, the attributes which all implementations of the ++VM must recognize, are displayed against a white background. Informational attributes, attributes which may provide additional information about an opcode's context but are not required for correct program behavior, are displayed against a gray background.

The virtual machine provides a number of built-in attributes to perform a number of different functions, including specifying opcode behavior, object lifetime, and method control flow. For the specifics as to which attributes are associated with which opcodes, see Appendix B.

Object behavior Attributes can specify additional information about the behavior of the opcode with which they are associated. For instance, some opcodes have attributes that specify that a certain object is frequently accessed and that the host machine should try to keep it in the cache. Other opcodes have attributes that specify that an object reference is unlikely to be null or that an object is likely to have a code pointer; these attributes help the virtual machine to predict object behavior and possibly eliminate unnecessary checks. These attributes can help the system make better use of the processor and the cache.

Object Lifetime Attributes can also specify information about the lifetime of an object or memory structure. For instance, opcode attributes can specify the length of time for which an object is expected to live so that an object can be put in the proper generation for generational garbage collection. Additionally, other opcode attributes can specify that an object should be allocated in a memory location such that it will

not conflict with another object. These attributes can help the virtual machine make better use of main memory and minimize the overhead needed by the allocator and the garbage collector.

Control Flow Attributes can also provide hints about control flow through a method. For instance, the opcode that throws an exception can specify a class in which it expects to find a handler. Alternatively, opcodes that rely on synchronization may use attributes to predict whether a block of code will be able to successfully grab a lock on the first attempt. These and other control flow attributes can be used to generate code that takes best advantage of the processor instruction stream and minimize the number of host processor cycles lost to code execution.

10.2 Features of Attributes

Attributes provide a number of useful features to the virtual machine.

Increased specificity Using attributes, ++VM instructions can be specifically tailored to a particular task. For instance, there is only one object creation opcode in the virtual machine; however, different attributes can be used to create a pinned object that will be around for a very long time. In this way, a small number of very general opcodes can be optimized for use in particular situations. Though the general opcode family remains the same, the attributes provide a more descriptive way of depicting how an object will be used by the virtual machine.

Improved performance Attributes can also be used to boost overall performance in two ways. One way to do this is to provide a series of hints to the just-in-time compiler in order to allow it to generate more efficient machine code. If the offline compiler is able to predict certain patterns in runtime behavior, the just-in-time compiler will not have to work as hard. However, the annotations provide more than just an offline method of providing information to a just-in-time compiler. Since the static compiler does not have to worry about the time taken to generate efficient code, it is able to perform more time-intensive improvements than may be possible at runtime.

Optional implementation Informational attributes are optional; the ++VM virtual machine will produce exactly the same result whether or not the suggestions

provided by the attributes are followed. Certain ++VM implementations may not choose to support every attribute, as they may suffer from size constraints or may lack hardware support for certain features and feel that it is inefficient to implement them in software. As long as the attribute information is provided, though, the virtual machines that choose to take advantage of the attributes may do so.

10.3 Annotation Opcodes

Like attributes, annotations are a way of providing additional static information to the runtime system. Unlike attributes, however, annotations are defined by a particular implementation of a virtual machine and may not be recognized by or supported on all ++VM implementations.

10.3.1 Annotation Layout

The annotation is divided into three parts: the annotation opcode, the annotation name, and the annotation behavior. Since all opcodes in the ++VM begin on word boundaries, the annotation attribute must begin and end on a word boundary. Thus, the annotation cannot contain a fractional number of words. Figure 10.2 shows an annotation as it would appear in the opcode stream.

Opcode The first word of the instruction is the annotation opcode. The first byte of the annotation opcode is defined by the ++VM specification to specify annotation opcode. These are the numbered bits in figure 10.2. The second byte holds the length of the opcode, if the opcode is less than 128 words in length, or an index into the constant pool otherwise. These are the letter bits in the first word of the annotation opcode in figure 10.2. The length of the annotation does not include the annotation opcode or the name index word.

Name The second word of the instruction is an index to the name of the annotation as held in the class's constant pool. Implementations of the ++VM should ignore annotations that they do not understand. If the annotation is unrecognized, then the virtual machine should skip the proper number of words in order to find the next instruction to execute. In figure 10.2, the I bits in the second opcode word specify the opcode name.

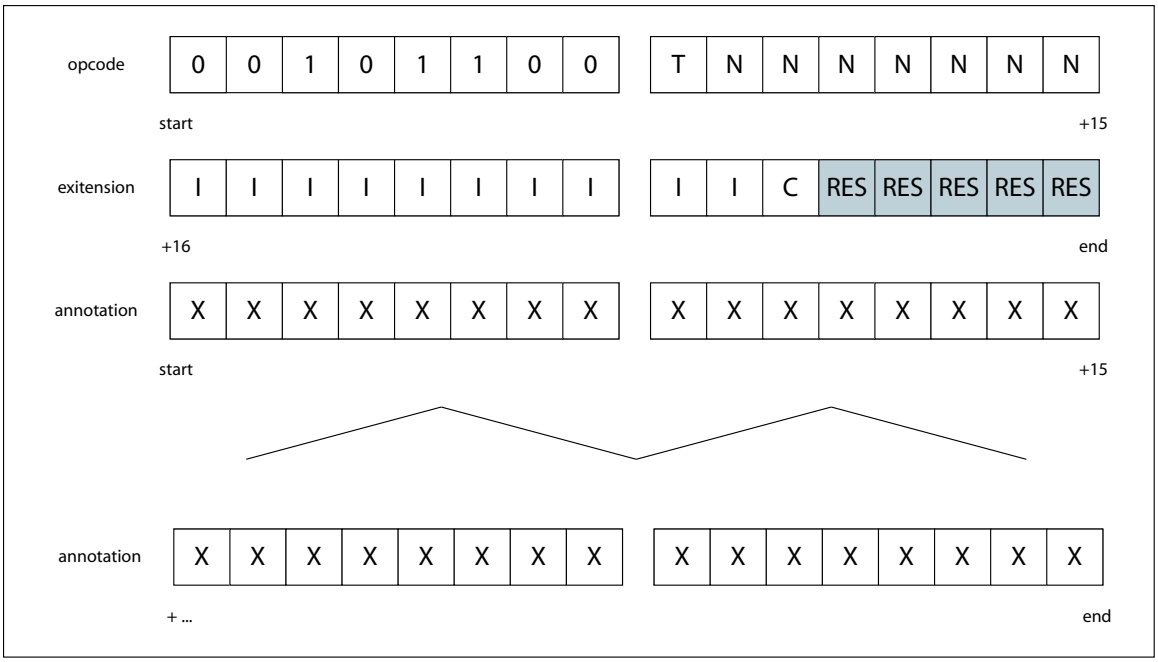

Figure 10.2: A ++VM annotation

Behavior The remaining words in the instruction are the body of the annotation. The behavior of the annotation is implementation-dependent, and its behavior must be defined by an agreement between the compiler writer and virtual machine implementer. In figure 10.2, these are the N bits in the remaining words.

10.3.2 Defining Annotations

Unlike all other opcodes and all other opcode attributes, annotations are specific to the implementation of the ++VM. An annotation is a contract between the code generator and the code user, a contract that specifies a particular kind of behavior that both the compiler writer and the virtual machine implementer wish to support. There are two steps for creating an annotation, getting the compiler writer and the virtual machine implementer to agree to an annotation, and getting both parties to use them.

Before an annotation can be used, both compiler writers and virtual machine implementations must agree to the form and behavior of an annotation. Since the general form of particular annotations are not enumerated in the language specifica-

tion, both compilers and implementations must agree to the internal specification of an attribute. More importantly, both groups must decide what the behavior of the annotation will be.

In order for an annotation to be used, it must be included in the virtual machine implementation by the implementer as well as inserted into the code by the compiler. Since individual implementers will choose whether or to support a particular annotation, not all annotations will be supported on all platforms. Additionally, since individual compilers choose whether or not to take advantage of a particular annotation, some compilers may not choose to create code which uses it.

10.4 Adopted Annotations

While most annotation opcodes are user-defined, some of the basic ++VM opcodes can be considered as common annotations that have been adopted into the language. These include the loop opcode and the block creation opcode.

Loops The loop opcode is an example of an annotation opcode. It defines a loop as a sequence of code blocks executed in a particular fashion, and provides a number of attributes that can allow for more efficient code execution. The loop instruction is an annotation because it only modifies the behavior of code execution, not the observable behavior of the program. For more information about loops, see Section 12.3.

Blocks The block creation opcode is another example of an annotation. It defines a block as a sequence of opcodes to be inserted at a particular location, and specifies a number of attributes about that section of code. As an instruction, however, it does not modify the state of the program; it only provides information about the behavior of code which is being executed. For more information about blocks, see Section 11.1.

10.5 Features of Annotation Opcodes

There are many reasons to use annotations when generating code. Code annotations will generally improve performance or help out the virtual machine in a variety of ways.

Improved Code Quality Many ++VM annotations will be geared towards increasing the quality of the generated code. There are several ways in which anno-

tations can take advantage of this. First, annotations can highlight specific features of hardware, such as parallel support or special processing units, that might not be available to standard machines. Additionally, annotations can give hints about runtime behavior of a program, such as additional information about object allocation or method invocation. By using annotations that take advantage of these features, the ++VM may be able to generate more efficient code.

Decreased Code Generation Time Some annotations may decrease code generation time. These annotations will be able to suggest optimizations to the JIT compiler which it would otherwise have to determine at runtime. Though these annotations only save time when the code is being just-in-time compiled, there may be opportunities where the time savings is substantial.

Increased flexibility Since the behavior of an annotation need only be defined for a single implementation, annotations are an appealing way for a programmer to quickly customize the ++VM to a particular application or architecture. Additionally, annotations provide a simple way for a group of users to quickly expand the instruction set. If an annotation becomes very popular, future design committees may be able to incorporate ideas from these attributes into the language.

Optional Use Certain ++VM implementations may choose to ignore some or all annotations, while others may implement a large number of them. They may choose to implement certain annotations to take advantage of hardware features such as parallel processors, or may choose not to implement an annotation for space reasons. Certain compilers and code generators may use annotations liberally or sparingly. High-end compilers may emit code that is specialized for specific tasks, while other compilers may not be sophisticated enough to gather information for a specific attribute. If both the virtual machine and the compiler choose to take advantage of annotations, the ++VM language provides the flexibility to improve performance; if either party decides not to, though, the language will continue to function.

Potential Disadvantages Though there are many advantages of using annotations, compilers and implementers should be aware of the potential drawbacks of using annotations. Two minor disadvantages are that annotations may result in potentially large increase in code sizes, and the fact that a potentially non-trivial amount of code may be ignored by most other virtual machine implementations. A more important

issue, however, is that the overuse of annotations may make certain code blocks heavily dependent upon specific virtual machine implementations; programmers must be especially careful not to use attributes to achieve performance targets. Despite these disadvantages, the ++VM chooses to implement annotations because of the many potential benefits that they can provide for future virtual machine designers.

Chapter 11

Code Groups

When high-level languages organize their code, they break the code up into sections in order for more efficient processing. Though many virtual machines break create these sections at runtime, these subsections can also serve as compile-time organizational units. Using a block structure would allow a compiler to group associated code units together, leading to a virtual machine that is more aware of the high-level behavior of its code and better able to optimize and execute code.

The ++VM supports several ways to group chunks of code together. First, the compiler can insert code blocks directly into the compiled code. These blocks can be used as ways to reduce code size, as ways to suggest potential code optimizations, or a combination of the two. Additionally, the programmer can reference pre-written code blocks stored in a library. These code blocks may perform a certain operation extremely efficiently, perhaps by running native code, and also provide a convenient interface for making use of previously existing code.

The remainder of this chapter discusses code groupings in the ++VM. It introduces the block structure and library structure, then discusses the various potential uses of each for writing efficient programs. It concludes with a discussion of the numerous performance and organizational benefits that code groups can provide.

11.1 Code Blocks

In the ++VM, code blocks provide a convenient way to group opcodes together and to provide additional high-level information to the just-in-time compiler.

11.1.1 Defining blocks

A block definition has three parts: the block header, the block body, and the block footer. A picture of a code block is shown in figure 11.1.

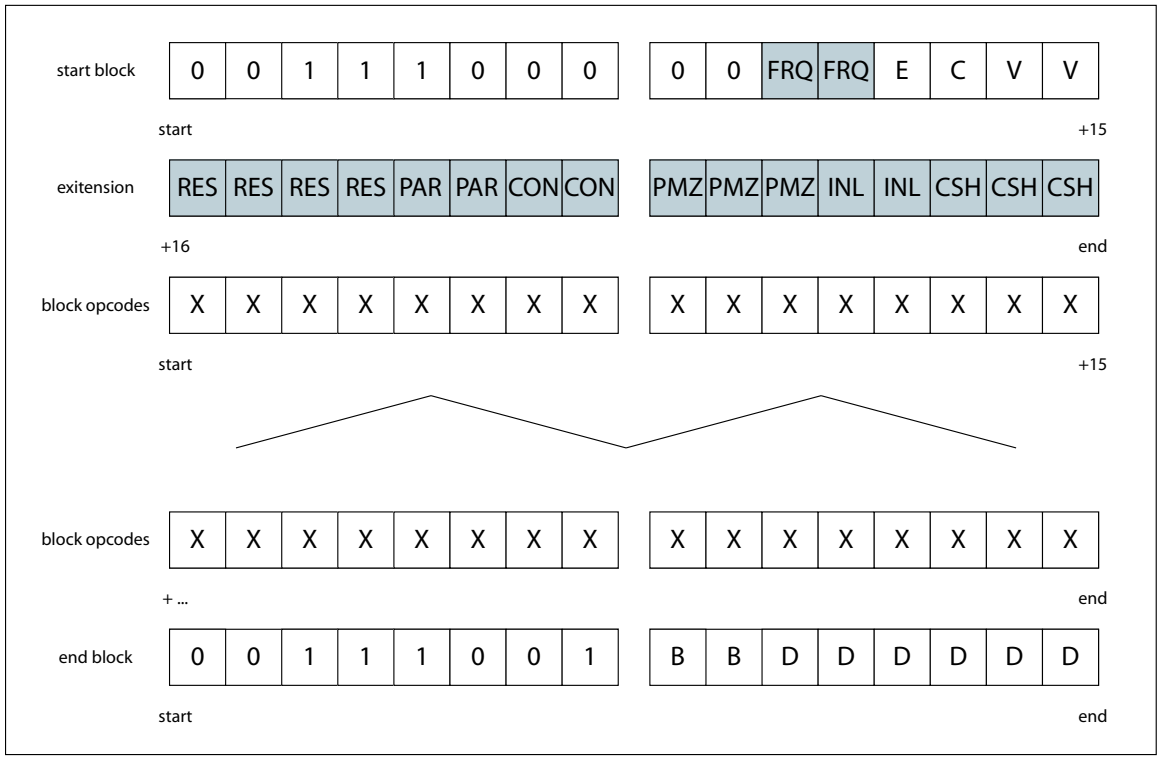

Figure 11.1: A ++VM code block

Header The block header consists of a single opcode, the block creation opcode. As shown in figure 11.1, the block header consists of the first opcode in the block and an additional extension word's worth of attributes. This opcode denotes a code block because the two most significant bits in the attribute word are 00, indicating that the remainer of the block will consist of ++VM opcodes. Block attributes can be subdivided into two sections: block attributes, attributes which convey information about how the block should be optimized, and opcode attributes, which convey information about the context in which the block will be used. These attributes are discussed in Section 11.1.2

Body The block body consists of the collection of opcodes which are associated with a particular block. Block bodies can be of arbitrary length and can make use of all intrinsic and extrinsic opcodes; a block body ends when the end block opcode is reached. Block bodies can reference, as well as define, other blocks, and are allowed to invoke library functions. However, blocks are not allowed to make circular references; a block cannot include any block which eventually calls itself.

Footer The block footer, the last opcode seen in figure 11.1, consists of the end block opcode followed by the opcode to which this block should be associated. The opcode to which the block will be assigned is a number between 64 and 255; the particular opcode chosen will affect the scope in which the block is visible. For more information on the opcode scope, see Section 11.3.

11.1.2 Block Attributes

There are six different types of block attributes in the ++VM.

FREQUENCY The two FRQ bits specify how frequently the block will be accessed and may provide information that is helpful to the cache. If the block is frequently accessed, the virtual machine should try to keep it in the code cache.

PARALLELIZE The two PAR bits specify whether the block can be parallelized over multiple threads. These bits can be used to encourage, discourage, or forbid the virtual machine from parallelizing block computation.

CONSTANT The two CON bits specify whether the inputs to the block are constant. If inputs are constant, the virtual machine may be able to perform a number of aggressive constant folding techniques.

OPTIMIZE The three PMZ bits specify an optimization level. This optimization level can vary from strict interpretation to complicated optimization, though increasing optimization levels may detract from performance since host instruction generation may take longer.

INLINE The two INL bits specify whether this block should be inlined wherever it is called. Inlining blocks can speed up code execution, though it can also cause code to rapidly expand in size.

CACHE The three CSH bits indicate another reference register with which this block will be frequently called. The virtual machine should place the block in such a way so that it minimizes cache conflicts with the reference in this register.

In addition to these attributes about general block behavior, each block can define up to eight attributes which are specific to a block. These attributes, the user-specified attributes, are equivalent to an eight-bit annotation that can be applied to a block. The user-specified attributes allow the programmer to provide extra information to the block when executing the block instruction. The implementation of these user-specified attributes is not defined and is left to individual virtual machine designers.

11.2 Accessing Libraries

In contrast to blocks, code groups that are written by the end programmer, libraries are written by the virtual machine implementer in order to extend the functionality of the ++VM. Linking an opcode to a library is an easy way to access additional features that are not available in the ++VM instruction set or to access pre-optimized code for a particular function.

11.2.1 Libraries

There are two types of libraries in the ++VM: standard libraries and utility libraries.

Standard Libraries The standard libraries contain ++VM instructions that link the virtual machine to the operating system and the hardware. These standard functions will all be implementation-specific. For instance, the standard libraries will have operations to print a line on the screen or to interact with shared memory in the system. Standard libraries must be provided along with all implementations of the ++VM virtual machine.

Utility Libraries The utility libraries contain instructions that are very similar to blocks. These instructions may be no more than a collection of opcodes linked together, but they can also be chunks of code that have already been optimized for a particular architecture. The utility libraries will contain useful data structures as well as functions; for instance, it might contain hash tables and string searching methods. While the presence of utility libraries will greatly enhance the performance of the ++VM, utility libraries are not required for the functioning of the virtual machine.

11.2.2 Binding Libraries

Library functions, like blocks, can be bound to opcodes. This is done using a special setting of the block opcode, as seen in figure 11.2.

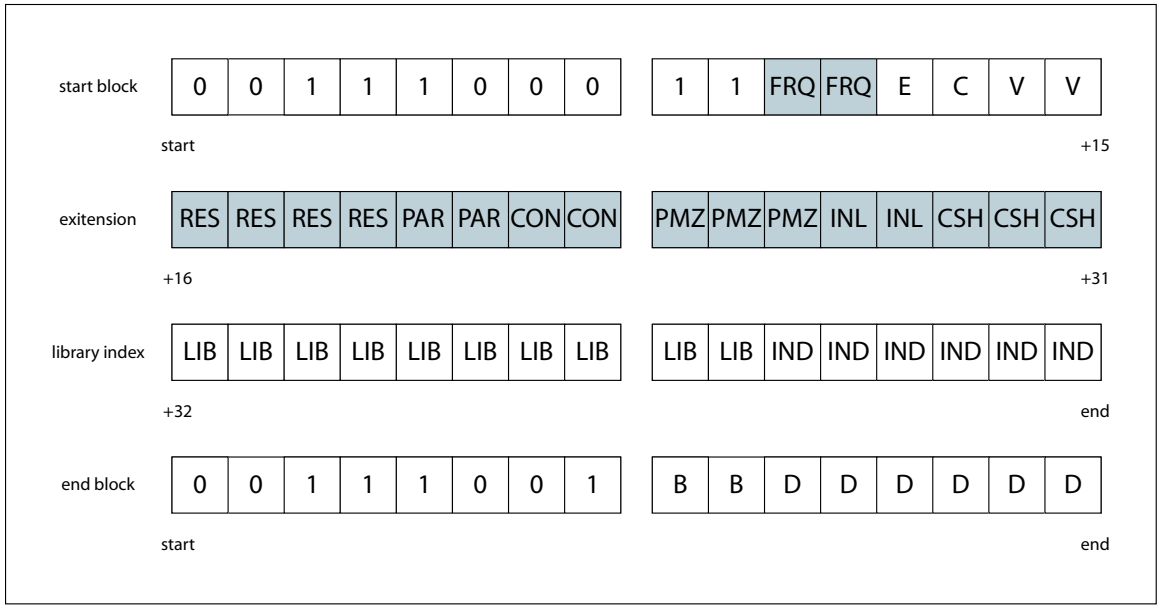

Figure 11.2: A ++VM library call

Header The library header consists of a single opcode, the block creation opcode. As shown in figure 11.2, the block header consists of the first opcode in the block and an additional extension word's worth of attributes. This opcode denotes a library function because the two most significant bits in the attribute word are 11, indicating that the remainder of the instruction will consist of machine instructions coming from a library. Library functions share the same set of attributes as blocks; more information about these attributes can be found in Section 11.1.2.

Body The library body in figure 11.2 consists of a call to a sequence of native machine instructions, though library functions also be calls sequences of ++VM opcodes instead.

Footer The block footer, the last opcode seen in figure 11.2, consists of the end block opcode followed by the opcode that this library function should be associated.

The opcode to which the block will be assigned is a number between 64 and 255; the particular opcode chosen will affect the scope in which the block is visible.

11.3 Using Code Groups

Blocks and libraries are used in the same way through the use of extrinsic opcodes.

11.3.1 Usage

A block or library is used by invoking the opcode to which it has been bound. The opcode instruction for a block must be between 64 and 255; the block must have been previously bound using the block mechanism described in Section 11.1. Blocks can have up to eight bits of user-specified attributes in the attribute byte of the block instruction.

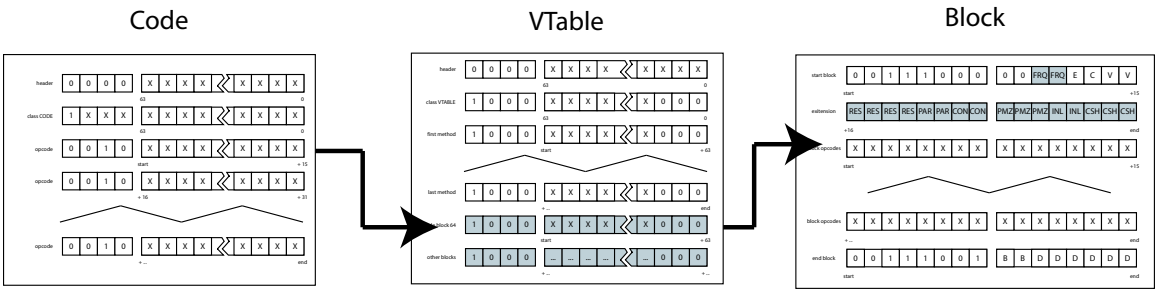

Figure 11.3: Block Use

Figure 11.3 depicts how the ++VM executes a block - in this case, a block bound to a particular method - if the block is not inlined. When the virtual machine encounters a block opcode or a library opcode while executing method code, it determines the location of the code bound to the block by checking the method vtable. The index in the vtable specifies the location of the code which can then be executed. However, if the block had been inlined through the use of the inline attribute, the block's code would have been inserted directly in the method when both were just-in-time compiled; no lookup would be necessary.

11.3.2 Scope

Before a block or a library function is used, it is necessary to verify that the particular opcode is in scope. There are three possible opcode scopes: per-method scope, per-

class scope, and per-thread scope. If a particular opcode is not defined, or is defined in the wrong scope, the virtual machine will throw an exception and must not execute the incorrect block.

Per-Method Blocks A class can define up to sixty-four unique per-method opcodes by using the extrinsic opcodes opcodes numbered between 64 and 127. Per-method blocks can only be used inside a particular method, and cannot be used by other methods in a class. One implementation for looking up blocks or library functions would be to attach them to the end of a particular method and to jump to the appropriate block accordingly.

Per-Class Blocks A class can define up to sixty-four unique per-class opcodes by using the extrinsic opcodes numbered between 128 and 191. Per-class blocks can be accessed in any method that the particular class defines, but cannot be accessed from outside of that class. One implementation for looking up the block or library function associated with per-class opcodes would be to look for these blocks an offset within the class vtable.

Per-Thread Blocks A class can define up to sixty-four per-thread opcodes by using extrinsic opcodes numbered between 192 and 255. Per-thread blocks can be accessed in any method of that thread; however, methods in other threads are unable to access these blocks. One implementation for looking up blocks with these opcodes occurs from an offset within the thread vtable. Opcodes defined in one thread cannot be used in another thread; the block or library must be defined separately in each.

Block Use Blocks should be placed in the minimal necessary scope. Thus, blocks should be defined at the method level wherever possible and only defined at the thread level when absolutely necessary. When a block must be used in multiple methods, a per-class opcode should be used; when used in multiple classes, a per-thread one should be employed. As there are only a small number of the larger-scoped opcodes, however, compiler writers should take great care in assigning to them.

11.4 Features of Code Groups

Grouping code, whether via code blocks or library functions, can provide a number of significant advantages, including the preserving high-level information, pre-arranging

JIT blocks, providing JIT hints, allowing for joint optimizations, supporting anonymous blocks, encouraging code reuse, and highlighting the parallels between code groupings.

Information Preservation Blocks and library functions provide an excellent way of conveying more high-level information directly to the virtual machine. If the blocks reflect the high-level structure of the code, then the optimizer will have a better picture of what the user's intentions and programming style. Using this information, the ++VM just-in-time compiler will be able to make more informed choices as to what kinds of optimizations to perform.

Pre-Arranged Blocks When a just-in-time compiler optimizes code, it does so by first breaking the code into sections. In the ++VM, the block structure of the code allows the static compiler to provide this information to the JIT rather than forcing the JIT to make potentially incorrect assumptions about optimization blocks.

JIT Hints Furthermore, this block optimization scheme allows for the static compiler to give hints to the just-in-time compiler. For instance, if a certain block will be executed multiple times, the static compiler can tell the JIT to heavily optimize a particular section of code without waiting for the JIT to discover this itself. Additionally, if a particular section of code will be particularly difficult to just-in-time compile, the static compiler may be able to give a precise estimate of its cost.

Joint Optimizations If blocks are nested within each other, the just-in-time compiler may be able to perform optimizations that cross block boundaries. For instance, it may be able to create a more favorable machine register allocation by having more information about the surrounding block.

Anonymous Blocks The ++VM provides some support for "anonymous" blocks by allowing blocks to be easily overridden. The opcodes in an anonymous block can be optimized according to the block attributes, but the optimizations will not be available in more than one location in the code. Anonymous blocks provide way of grouping opcodes together so that the JIT compiler can recognize the high-level structure of the code, but without tying down one particular opcode for an entire method.

Code Reuse These code grouping schemes highlighted in this section encourage the principle of code reuse. If a section of code is repeated multiple times, it may be worthwhile for a compiler to pull it aside once and use an opcode that references it multiple times. Since block nesting is allowed, compilers may be able to using nested blocks to further decrease the size of the code. As an added advantage, this decrease in code size will not lead to a decrease in runtime performance if the just-in-time compiler can inline the code.

Code Group Interface By treating all code groups in a similar fashion, the ++VM highlights the symmetry between code blocks and library functions. The common syntax comes as a result of common behavior: both blocks and libraries perform similar functions, and the virtual machine should perform the same tasks in order to support both of the. For instance, the just-in-time compiler can attempt to perform the same types of optimizations, including inlining and code manipulation, to library functions as well as blocks. By treating both code groups in similar ways, the ++VM highlights the similarities between them.

Chapter 12

Control Flow

A program is more than a linear sequence of commands. Program execution jumps around using various devices to control code execution. From Lisp to Java, past virtual machines have used a number of opcode-level mechanisms to channel execution within a method, including logical branches and exception handling. However, controlling program execution in a high level language can be very expensive. Just-in-time compilers have attempted to improve upon the performance of these mechanisms to keep the host processor busy executing new instructions instead of idling and waiting for a failed branch prediction to load from memory. Using JIT techniques, high-level languages have been able to greatly improve program performance.

The ++VM attempts to improve upon past work with virtual machines and further increase the speed of execution flow; it attempts to integrate the virtual machine more closely with hardware and with the cache in order to improve overall code execution speed. The ++VM accomplishes this in three ways. First, it uses built in branch prediction for logical jumps and also provides opcode-level support for futures. This allows for a more flexible method for changing control flow within a method. Second, it uses annotation opcodes to indicate the presence of loops and other repeated expressions. This allows a JIT to have a better picture of the high-level code layout so that it can perform more efficient optimizations. Lastly, the exception mechanism can be annotated to provide better performance. This allows the just-in-time compiler to understand how the exception mechanism will behave during code execution and allow it to optimize certain code sections accordingly.

The remainder of this chapter is organized as follows. The first part of this chapter explains how the ++VM supports logical control and the features of this approach. The next part introduces loop structures and discusses the benefits of using loop structures. The chapter concludes with a discussion of exceptions in the ++VM and

the features that this exception mechanism provides.

12.1 Logical Control in the ++VM

The most common manner of transferring control is via a local jump, a process that changes the program counter value possibly based on a condition. All local jumps are controlled by the values in the condition code register. For more information about the condition code register, see Section 6.1.3.

12.1.1 Comparisons

In order to branch, the bits in the condition code register must be set appropriately. This can be done implicitly, as the result of an instruction writeback, or explicitly, as a result of a comparison operation.

Implicit Writeback Every time the virtual machine performs an opcode operation, it writes back the condition codes corresponding to the operations result. Upon the completion of an instruction, the condition code register is set to reflect that result, allowing the next instruction to use the condition code bits in order to control program execution. In a sense, every operation performs an implicit comparison operation because all operations will result in changes to the condition code register.

Explicit Writeback The ++VM has several instructions to perform explicit writeback operations. These instructions are designed to prepare the condition code register in order to facilitate a branch. The compare data values instruction, CMD, sets the condition code bits based upon the comparison of two data values. The compare reference values instruction, CMR, sets the bits based on two reference values. The compare within bounds instruction, CMB, is a three-operand instruction that tests whether one data value is between two other values. These instructions should be used sparingly, as ++VM code should rely on implicit writeback wherever possible.

12.1.2 Branching

The ++VM has four kinds of jump statements: jumps on current values, jumps on initialization and future bits, jumps on other condition control register bits, and jump on previous values bits.

Current Values The virtual machine has an opcode for branching on the negative (n), zero (z), and overflow (v) bits of the condition code register. These bits are the standard bits used for most logical jumps.

Exceptions The virtual machine has a setting to branch if the exception bit is set. Additionally, there are settings to branch if an arithmetic overflow or arithmetic underflow occurred. This can be used to ensure that all values in the virtual machine are in a consistent state and that information is not accidentally lost.

Futures The virtual machine has a setting to branch based on whether or not the current value is a future. This can be used to ensure that operations are not performed on values which have not yet been computed.

12.2 Features of Logical Control

The branching scheme as outlined above allows for a more flexible and versatile manner of logical control. This, in turn, can give rise to many features that are not present in other virtual machines.

Branch Prediction All branch instructions in the ++VM have an inherent predictive element. For instance, a branch on equals opcode will prepare the cache and instruction stream in a manner that assumes that the branch will succeed. If the branch is not likely to succeed, the static compiler should rewrite the instruction. When the just-in-time compiler is run, it may be able to infer more nuanced information about the likelihood of a particular branch; the opcodes provide only rough information as to whether or not the conditional will succeed.

Bounds Checking The ++VM is able to perform bounds checking using the condition code register. The bounds opcode compares three numbers to determine whether the number being tested is in between a lower and upper bound. If the bounds test succeeds, it sets the z bit in the condition code register, signaling equality; if it fails, it sets the other bits appropriately. The bounds test can be followed by a conditional jump to take advantage of this information. Bounds testing can often be done efficiently in hardware, and the ++VM attempts to mimic that simplicity and possibly achieve better performance.

Pipelining By keeping track of the penultimate values in the condition code register, the virtual machine is able to potentially improve pipelining within the processor. Failures in instruction prediction can be potentially costly, as it may be necessary to wait for a processor pipeline to refill with instructions from the alternate pathway. By allowing branches on the penultimate instruction, the ++VM provides a mechanism for potentially minimizing the number of incorrect branch predictions: once a test is run, the virtual machine executes an intermediate instruction while it decides which instruction to load in the pipeline.

Performance Condition code registers are extremely common in processors, as some form of condition code register is commonly used for branching at the hardware level. However, condition code registers are rarely found in virtual machines: languages from Smalltalk to Java prefer to push and pop conditional arguments from the stack and branch accordingly. The use of a condition code register will likely result in an overall performance increase. For one, the use of a register to store condition code information will save the virtual machine from pushing and popping stack arguments. Additional performance increases can be realized if the condition codes are directly supported by the host's hardware, as often may be the case.

Skimping When virtual machines are implemented completely in software, it may be possible to omit certain updates of the condition code register in order to improve performance. During code execution, the virtual machine will be able to identify blocks of code which do not immediately precede logical jumps. In these areas, the virtual machine will be able to avoid updating the condition code register as those bits will never be used by a jump instruction. As long as the virtual machine is able to properly reconstruct the contents of the condition code upon an exception or an internal error, a software implementation of the ++VM is free to take shortcuts and make optimizations to improve performance.

12.3 Loops in the ++VM

In the ++VM, loops are implemented using an annotation specifically designed for loops. This annotation helps to convey information about the loop to the just-in-time compiler and to the cache. The loop opcode itself is not interpreted by the virtual machine; instead, it signals the boundaries of the different blocks that make up a loop.

12.3.1 Using Loops

Loops in the ++VM are divided into four component blocks, as detailed in figure 12.1.

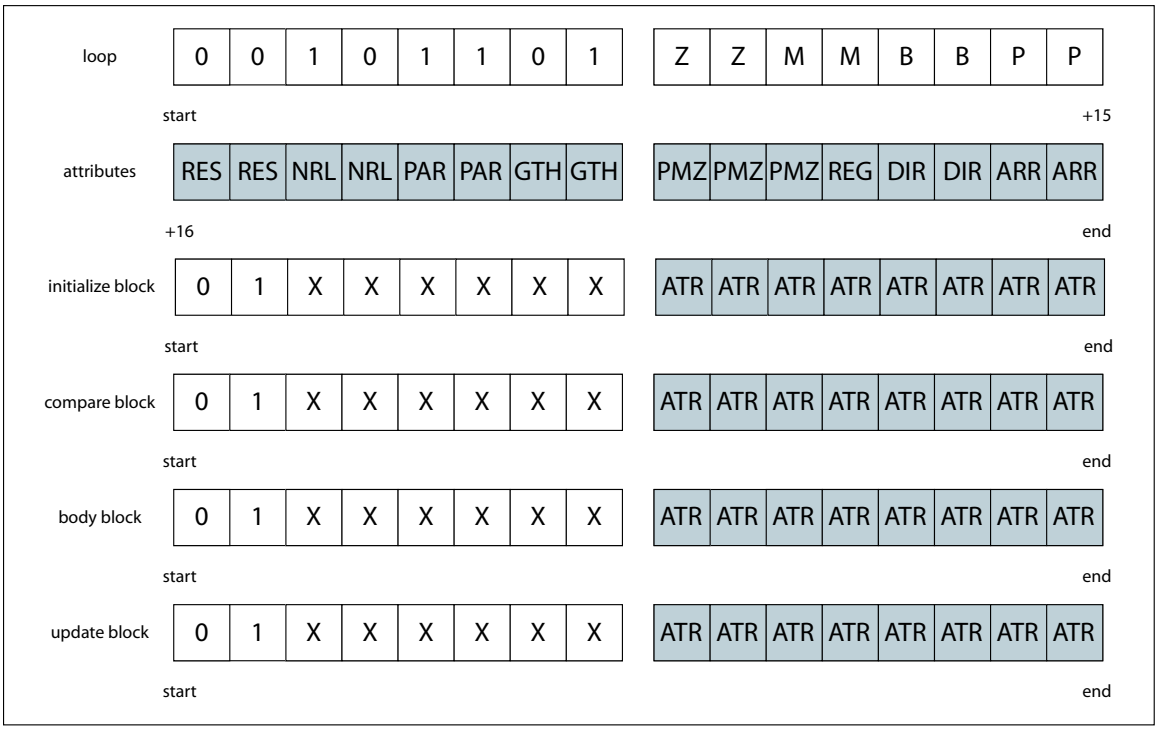

Figure 12.1: A ++VM loop

12.3.2 Loop Components

Loops consist of four component blocks: the initialization component block, the comparison component block, the body component block, and the continue component block. Though loop component blocks may be code blocks, they do not have to be.

Initialization The initialization block, specified by the ZZ bits in figure 12.1, is executed on loop entry. The ZZ bits specify the method of determining how to find the initialization block in the opcode stream. It contains code to set up variables that are in scope within a loop and to initialize those variables to the proper value. The initialization block will only be executed once before execution proceeds to the loop body.

Comparison Block The comparison block, specified by the MM bits in figure 12.1, is executed each time through the loop. The MM bits specify the method of determining how to find the comparison block in the opcode stream. If the comparison succeeds then the loop body is executed, otherwise execution will continue with the code that follows the loop. An attribute can be set to skip the comparison block immediately after the initialization block; this will allow for the representation of a do-while loop.

Loop Body The loop body, specified by the BB bits in figure 12.1, contains the instructions that will be executed repeatedly until the expression in the comparison block is false. The BB bits specify the method of determining how to find the loop body in the opcode stream.

Continue Block The continue block, specified by the PP bits in figure 12.1, holds code that is executed every time execution successfully reaches the bottom of the loop body. The PP bits specify the method of determining how to find the continue block in the opcode stream. It can perform modifications to a loop counter or some other common iteration technique. The continue block can be explicitly skipped using instructions that branch back explicitly to the comparison block.

12.4 Features of Loops

The ++VM gains several advantages by representing loops in this fashion. As current virtual machines do not have loop opcodes, they are not able to take advantage of these features.

Loop Attributes In the ++VM, loops can have a variety of attributes. For instance, it is possible to specify whether a loop should be unrolled, whether a loop should be parallelized, or even whether the loop counter will move in a predictable fashion. The ++VM virtual machine can use this information to speed up computation, to move computation around between different threads, and to manage the cache as efficiently as possible.

Loops as Code Blocks Conceptually, loops and code blocks are very similar: in the same way that a code block brings together a group of opcodes together with attribute information, a loop brings together four blocks with annotation information.

Both loops and code blocks provide additional information about code behavior in order to generate efficient JIT code. For more information about code blocks, see Section 11.1

Loop Components as Code Blocks Every loop component can be a code block, a feature with two benefits. First, code blocks can be optimized individually. Additionally, though, blocks can mark loop boundaries so as to give hints about how the loop control structure will perform in aggregate. By first presenting the component blocks individually, then presenting the code together, the layout of the loop opcode provides a suggestion as to how the loop should be optimized.

Caching and Prediction The loop opcode provides support for caching and branch prediction in ways not found in other virtual machine instruction sets. Other languages do not provide explicit hints as to different sections of loops, and instead rely on a series of branches in order to establish loop behavior. The ++VM, on the other hand, will be able to determine which loop block will be executed next, allowing the virtual machine to prepare the cache more intelligently.

Parallelization The loop opcode allows the compiler to convey information about if or how the loop can be parallelized. If statements in the loop body do not have side effects, the virtual machine may be able to spawn new threads - or ship loop sections off to other processors - for faster computation. If the compiler can provide this information at compile time, the virtual machine will be more prepared to make high-level decisions about how to execute the loop and what resources the loop should be given.

Loop Counters Though the static compiler initially assigns registers, the virtual machine may choose to reassign them at runtime. The loop opcode, however can tell the virtual machine whether or not the loop counter register should be permanently assigned to a register because it will be commonly used and commonly accessed. For instance, in small loops it is essential to reserve a register for the loop counter rather than copy it back and forth from memory. This assignment can improve overall performance.

12.5 Exceptions in the ++VM

The virtual machine has two ways of throwing exceptions as well as two ways of handling exceptions.

12.5.1 Explicit Exceptions

An exception can be thrown explicitly using the TRW instruction, as shown in figure 12.2. Explicit exceptions are created as instances of a subclass of the class Exception; the particular class of the exception is specified in the constant pool of the throwing class. The exception to create is specified by an index into the throwing class's constant pool. If the index can be represented with five bits or less, the entire throw instruction fits onto a single word; otherwise, an additional extension word is needed to represent the entire ten-bit address.

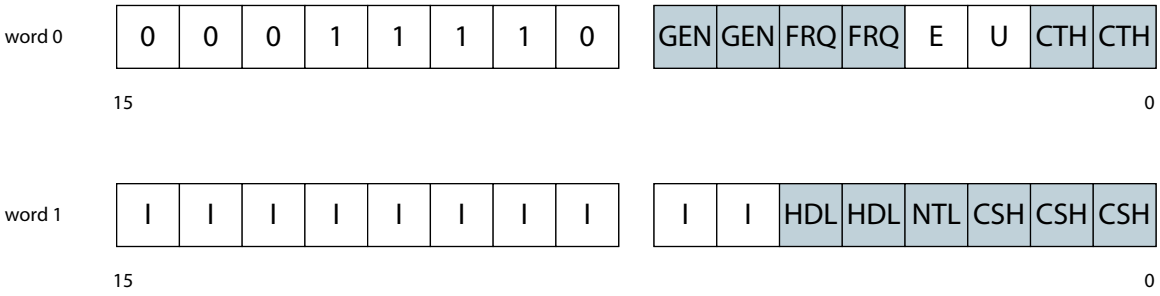

Figure 12.2: The Throw Opcode

Explicit instructions can have several attributes.

GENERATION The GEN bits specify how long the exception will live, and provides a hint to the virtual machine about garbage collection. Increasing levels provide information about how long the exception will live; only in rare cases will it be necessary to state that exceptions will live for a long time

FREQUENCY The two FRQ bits specify how frequently the exception will occur. From this frequency information, the virtual machine can decide which exception handling scheme to use

CATCH The two CTH bits specify where the exception will likely be caught. This information can be used to jump-start a search for an appropriate exception handler

HANDLE The two HDL bits specify how the exception will be caught. This provides some prediction information about the behavior of the handler and may speed up exception throwing by allowing the virtual machine to eliminate unused information

INITIALIZE The NTL bit indicates whether this object should be initialized. If the exception handler uses only the class of an exception object without accessing any fields in the exception itself, then the exception initialization method should be skipped.

CACHE The three CSH bits indicate another reference register with which this exception will be used. By allocating the exception intelligently, the virtual machine may be able to speed up code execution.

12.5.2 Runtime Exceptions

The virtual machine will also throw exceptions on its own accord if opcode parameters is malformed; these exceptions are runtime exceptions. For instance, it will throw an arithmetic exception if the virtual machine attempts to divide by zero. The virtual machine will also throw exceptions if casts or conversions fail, or if the operands to a particular instruction are not of the same type. Runtime exceptions are handled using the same exception handling scheme as is used for all other exceptions.

12.6 Handling Exceptions

The ++VM has two methods of handling exceptions, stack unwinding and stack cutting.

12.6.1 Stack Unwinding

Exception handling via stack unwinding, the default handler for high-level languages like Java, is most efficient in cases when exceptions are rare. When an exception is thrown, the virtual machine looks for a handler in a side table. The program counter is used to search in an exception table to determine the appropriate handling code; if an exception handler is not found in the current method, its frame is popped off the stack and the search process is repeated in caller methods.

When using stack unwinding, there is no penalty in the expected execution path. Though looking up an exception is a relatively inefficient process, it does not require

any setup. When using stack unwinding, it is not necessary to use an opcode to set up a try-catch block; the try-catch block is handled in a side table. Additionally, this scheme has the advantage that the side table can be kept separate from the primary opcode stream, meaning that it does not have to be loaded into cache until an exception is thrown.

12.6.2 Stack Cutting

Exception handling via stack cutting, used in high-level languages such as Perl6, is most efficient when exceptions are relatively common. Before entering a block that can throw an exception, the virtual machine explicitly loads an exception handler. Typically, the handler loading process involves saving the type of exception in a list of try blocks; in the ++VM, this is implemented by saving the exception handler in a stack held in a dedicated thread register, the exception register. When an exception is thrown, the virtual machine jumps to the appropriate block using the exception register as a guide as to where to handle the exception.

When stack cutting is used, there is a substantial penalty in the expected path. Every time a try block is entered, it is necessary to change the value in the exception register. Additionally, every time execution leaves a try block normally, it is necessary to remove that exception from the exception register and restore the old value. When exceptions are rare, this involves a substantial amount of overhead that could be avoided by using stack unwinding.

12.6.3 Finally Blocks

Unlike many other high-level language virtual machines, the ++VM does not provide direct support for finally blocks. A finally block is a block of code that will be appended to every possible pathway through the exception handler, whether that pathway contains an exception or not. As finally blocks cannot be implemented easily and efficiently in hardware, they are not included in the basic specification of the virtual machine. Language designers who wish to use the ++VM may choose to implement finally blocks in a supplemental instruction set. It is expected that an implementation of finally blocks will probably make its way into the standard libraries that come along with the virtual machine.

12.6.4 Errors

The ++VM uses the exception-throwing syntax to throw virtual machine errors. However, because errors are unrecoverable problems within the virtual machine, they cannot be caught and will cause the virtual machine to shut down.

12.7 Features of the Exception Mechanism

The implementation of exceptions in the ++VM allows for dynamic handler changing, exception attributes, and handler attributes.

Dynamic Handler Changing The ++VM has two exception handlers, one that uses stack cutting and one that uses stack unwinding. When an exception handler is created, the user or the compiler can specify whether the exception is likely to occur or not. If the compiler or the user is able to predict that exceptions are likely to occur, the virtual machine can use the JIT to implement a stack cutting mechanism which catches exceptions efficiently. On the other hand, if the compiler is able to predict that exceptions are rare, the JIT should use a stack unwinding strategy. Because the virtual machine can dynamically change its exception-handling strategy based on attributes in the opcode stream, it should be able to handle exceptions much more efficiently than if it were required to commit to an exception handling strategy before commencing code execution.

Exception Attributes The exception throwing opcode provides a number of attributes which can help the virtual machine support exceptions more efficiently. Coupled with an effective error handling technique, this will help improve overall performance.

Handler Attributes The handler setup opcode provides a number of attributes which can help the virtual machine decide on an appropriate exception handler and to handle exceptions efficiently. For instance, the handler opcode can provide information about exception frequency, handler lifetime, where a caught exception will be thrown, and the specific exception that is being caught. All of these properties can help increase overall performance by giving the virtual machine more information about how the code operates.

Chapter 13

Conclusion

The science of constructing high-level language virtual machines can no longer be treated as an afterthought in language and hardware design. On the one hand, the current increase in the number of virtual machines can be seen as a reaction to the lack of improvements in modern hardware. Language designers want certain features today, and hardware designers have not provided them. On the other hand, hardware designers have found that virtual machines provide a convenient escape mechanism to avoid implementing abstract concepts in silicon. Hardware designers can provide efficient support for high-level code, but only if they have a firm understanding of what that code is doing. The ++VM virtual machine attempts to address these desires by creating a modern virtual machine that supports contemporary, and future, programming styles.

13.1 Features of the ++VM

Two common criticisms of virtual machines is that they are slow and that they are unnecessary. Neither of these need be the case. Virtual machines, and the ++VM especially, are now beginning to address both of these concerns. The preceding chapters have introduced a number of features that convincingly counter these criticisms. While the advantages of these features have been described in the context of the ++VM virtual machine, they are applicable to other virtual machines as well.

Speed Virtual Machines do not have to be inefficient. Though historically languages running on virtual machines have been much slower than those compiled to machine code, some of this performance hit has been due to intentional design decisions. This

need not be so. Virtual machines like the ++VM should be forward-thinking in terms of hardware and software: since they are at the intersection of hardware and software design, they can help both groups develop new and more efficient computation techniques. The ++VM achieves more efficiency in a variety of ways. First, ++VM instructions are intended to mesh well with current hardware. By supporting registers, providing a mechanism for interacting with the cache, and giving as much information as possible to hardware branch prediction mechanisms, the ++VM works in conjunction with the hardware to improve better performance. Additionally, the ++VM provides a powerful mechanism for hints to the just-in-time compiler. By using attributes and annotations, users can increase the performance of the JITed code as well as decrease the amount of time that the virtual machine must spend creating it.

Attributes Currently, hardware manuals and virtual machine specifications only specify required bits. This need not be the case: the ++VM introduces a system by which compilers and language designers can provide additional information to the virtual machine, but the machines can either take or leave this information. This allows implementers to decide what forms of information are important to the operation of a machine and what forms of information are superfluous. Certain ++VM attributes may be helpful to all implementations, such as attributes that specify the types of objects with which an object will be used; other attributes might not be so useful, such as generational garbage collection attributes on implementations that choose to use a mark-sweep collector. Optional attributes allow compilers to specify as much information as they desire while allowing an implementation to ignore this information and still function correctly

Annotations Machines, like humans, should comment their code. When a machine executes a sequence of low-level instructions, it may not be able to determine the context of these operations without some form of reflection. By providing a means of providing annotations or attributes to the code, machines like the ++VM will be able to understand the larger environment of the instruction. Using annotation information, a machine can perform a variety of optimizations to better support the operation. The ++VM supports a variety of annotations: in addition to opcode attributes and annotation blocks, it supports memory tagging, which provides information about the layout of memory, as well as object headers, which can store information about the behavior of an object. These allow the virtual machine to have an annotated picture of its memory space and may help it optimize memory

use. Additionally, annotations serve as a communication link between the platform-independent opcode stream and the host processor: as many processors reorder or modify their operations. The ++VM annotations assist this reordering process by providing the host system with additional information about the global state of the ++VM user program. Lastly, annotations can be used to directly influence the execution mechanisms of the virtual machines. Annotation information can be used to enable certain features of the virtual machine, such as exception handling mechanisms or generational garbage collection, or can be used by the just-in-time compiler to develop a more accurate picture of how the code should be executed.

Opcode Layout Virtual machine designers have leeway to introduce novel features to the layout of their instruction set. While past virtual machines have sought to lay out code in a platform-independent manner, virtual machines should consider ways of laying out code to improve efficiency. The ++VM introduces a number of new ways of looking at code layout issues. For one, the ++VM provides the block mechanism, a way in which a commonly-used string of opcodes to be available everywhere but stored in one location. This facilitates the compression of offline code, helps preserve some of the high-level structure of the source code, and provides hints to the just-in-time compiler as to the boundaries of a program's basic blocks. Lastly, the ++VM code annotation mechanism provides meta-information about code execution, a form of commenting that a virtual machine may be able to understand. Code annotations convey information about the instructions, and can make the code more specific, more informative, and faster.

Polymorphism Machines should be designed to be as flexible as possible in order to support various data types, and virtual machines are in an enviable position to be able to exploit this using polymorphic instructions. The ++VM uses polymorphism in several ways. First, the ++VM chooses the specific machine instruction to execute based upon a value's memory tag. For instance, math instructions and memory movement instructions are defined generically; the specific instance of the polymorphic operation is chosen based on the runtime data type. Furthermore, the ++VM attempts to highlight the conceptual similarity between many of the operations that it implements. For instance, the virtual machine uses the same opcode family for code blocks and library functions, and the same opcode family for creating objects and arrays, because they are very similar from the programmer's point of view. Lastly, the implementation of polymorphism on the ++VM frees up a number of bits in each opcode instruction as well as a number of opcodes in the instruction set. Because

of the benefits provided by opcode polymorphism, the virtual machine can support other important opcode features such as instruction attributes and blocks.

Object-Oriented Design Since many programmers use in object-oriented languages, virtual machines should continue to provide strong support for objects. Object-oriented virtual machines need not be inefficient and slow, and future virtual machines can address these performance issues with changes to how objects are treated. The ++VM attempts to make object-oriented design an integral part of the virtual machine while also improving performance. For example, objects in the ++VM are treated as primitive types as the instruction set has direct support for object creation and access. Object-oriented design has long been an important language feature and the ++VM, and the ++VM includes strong support for objects in the hope that future hardware designers will create object-oriented hardware to support similar virtual machines in the future. Additionally, it uses several tag types to differentiate between different kinds of objects. These tags can help improve virtual machine efficiency by providing object-specific code and avoiding unnecessary pointers. Furthermore, the ++VM specification defines the layout of several object headers, including those for list elements and classes. By choosing to implement these in a specific way, the virtual machine implementation is more uniform and can integrate more efficiently with external code sources.

Extensibility Virtual machines are much easier to extend than hardware. Hardware architectures are rather brittle and can require a large design cycle to change; in contrast, virtual machines can be modified, tested, and distributed - or scrapped - in a much shorted period of time. There are a number of ways to extend the ++VM virtual machine, ways that can be used to add new functionality or to improve existing features. For instance, the block and library mechanisms provide ways to add new functionality to the ++VM by tailoring a method's opcodes to the method's needs. The ++VM can be quickly customized for the behavior of a particular method by loading a new library function or binding a particular library block to a particular instruction. Additionally, the annotation procedure makes it easier to tailor a virtual machine to a specific task by adding additional context information. Annotations can be added or ignored at will, allowing the virtual machine to expand in scope if the user wishes to take advantage of the information provided.

Integration Virtual machines are designed to be easily modified to keep pace with hardware development. As the functionality of hardware increases, the need for a

large virtual machine in software diminishes. Over time, certain virtual machine features should be shifted from being purely emulated in software to partially or fully supported in hardware. Compared to compiled machine code, virtual machines can be much more easily changed to take advantage of these changes as more and more capable hardware becomes available. The ++VM is well poised to take advantage of this. Many of its features, including tagged memory, condition code register support, and garbage collection hooks, are designed to be easily integrated into future hardware. As more complex architectures are developed in the future, more and more ++VM components can be implemented in silicon, decreasing the overall complexity of the software component and improving performance.

Hardware Prototyping Virtual machines can also be used effectively to prototype new hardware systems. Rather than writing a complete compiler for a new target architecture, a hardware designer need only write an implementation of a thin virtual machine in order to perform tests and can decrease the time necessary to roll out new hardware and increase the overall pace of hardware development. The ++VM is especially useful in this regard, as it may be easier to implement some features of the ++VM virtual machine, such as an opcode interpreter or the garbage collector subsystem, than it would be to develop multiple compilers to support multiple languages. Hardware architects may find it simpler to implement these smaller sub-systems of the ++VM rather than changing the back-end of a compiler to implement and test programs on a new type of machine.

13.2 Future Work

There are a number of ways in which the ++VM and other virtual machines can be improved, including sample annotation creation, thread support, library definition, and mapping languages and implementation.

Sample Annotation Creation The ++VM is among the first systems to introduce a annotation system. Compiler writers may not know how to use this functionality, and it may be necessary to write a number of sample annotations in order to demonstrate how these annotations should be used and how they can improve code performance in the ++VM. Additionally, those who wish to extend the power of this virtual machine, without directly modifying the instruction set, may wish to provide a powerful suite of annotations for the ++VM opcode set.

Threading The current ++VM specification hints at, but does not directly address, the threading and scheduling components of the virtual machine. As parallel and concurrent programs become more prevalent in the future, these areas will become increasingly important as virtual machines like the ++VM will need to efficiently support multiple execution streams. Those who wish to extend the ++VM may wish to formalizing the thread and lock specifications, specify the thread swapping and priority mechanisms, and examine the effectiveness of the suggested thread synchronization mechanisms.

Library Definition Though the ++VM opcode set may be able to provide more functionality than opcode sets of other languages, it is still necessary to define important native methods that must be in the core libraries. While some of these functions are intuitive, such as methods for printing out strings or opening files, there may a number of subtleties in filling out the libraries. Future work on the ++VM may include codifying the types of library functions necessary to support the virtual machine.

Mapping Implementation As of this writing, the ++VM has not been implemented in software or in hardware. Some ideas suggested in this thesis have been shown to be individually effective but have not been combined in a project of this scale. Other ideas introduced in the ++VM are speculative, and there are no guarantees on how well the virtual machine will perform in practice. Future language designers may wish to prototype sections of or the entirety of the ++VM in order to determine whether these design decisions provide expected performance increases.

Mapping Languages Another area of future work is the mapping of languages to the ++VM. It is expected that many high-level languages will easily map to the ++VM, including such diverse languages as Java, Haskell, and Smalltalk. It is also expected that this virtual machine will be able to support low-level language programming for languages such as C. However, none of these claims have been officially tested. Those who wish to continue working with the ++VM may wish to implement some of these languages, or even entirely new languages, using the ++VM opcode set.

Appendix A

Tags

The first part of this appendix section specifies the data and reference values possible in the ++VM. It also presents the tags associated with these values. For more information about tagged memory, see Chapter 7.

How To Read This Section

Title The name by which the tag is called.

Figure A picture of how the tag is laid out in memory. The four bits to the right specify the tag.

Description A description of the data or reference type.

Refers A picture of the reference value to which the reference type refers.

A.1 Byte

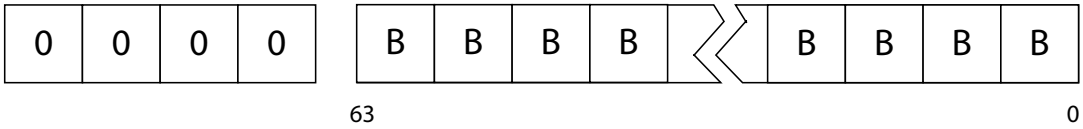

Figure A.1: Eight Bytes and Tag

This tag indicates that the associated sixty-four B bits are divided into eight eight-bit bytes. The bytes are unsigned.

A.2 Short

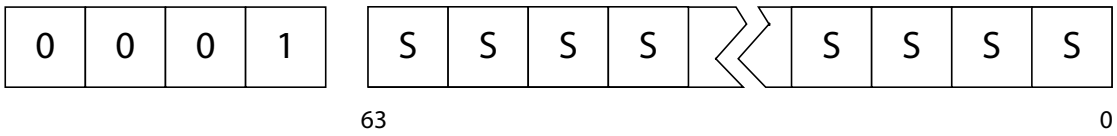

Figure A.2: Four Shorts and Tag

This tag indicates that the associated sixty-four S bits are divided into four sixteen-bit shorts. The shorts are unsigned.

A.3 Word

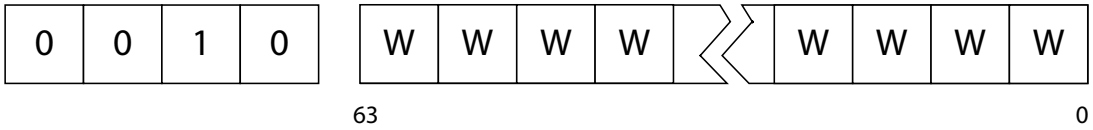

Figure A.3: Two Words and Tag

This tag indicates that the associated sixty-four W bits are divided into two thirty-two-bit words. The words are signed.

A.4 Long

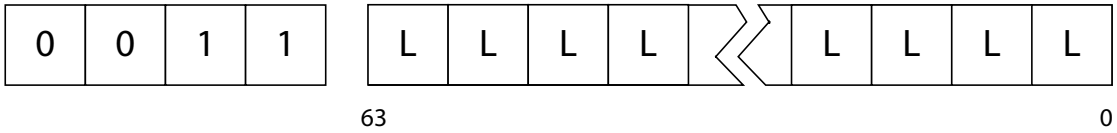

Figure A.4: Long and Tag

This tag indicates that the associated sixty-four L bits are one long word. The long word is signed.

A.5 Ultra

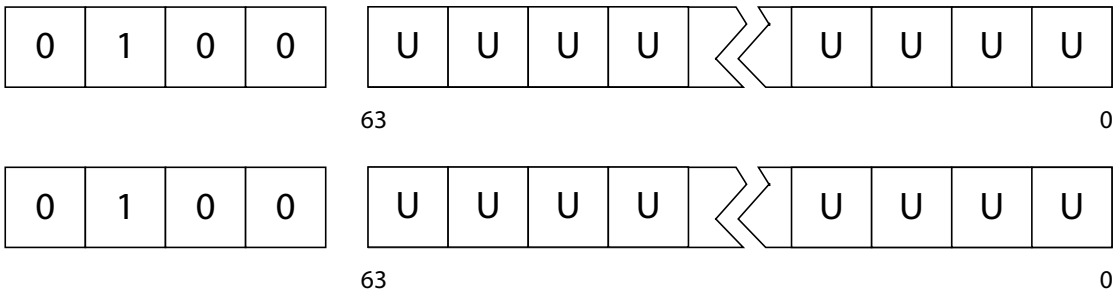

Figure A.5: First Word of Ultra and Tag

This tag indicates that the associated sixty-four U bits are half of an "ultra long" memory value, a signed 128-bit integer stored in little-endian format. If the sixty-four bits begin at a memory address that is divisible by sixteen, then the bits are the most significant bits of the ultra; otherwise, they are the least significant bits. The ultra is signed.

A.6 Float

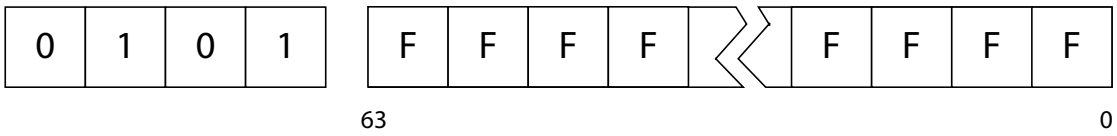

Figure A.6: Two Floats and Tag

This tag indicates that the associated sixty-four F bits are divided into two thirty-two-bit IEEE 754 floating point numbers. The floats are signed.

A.7 Double

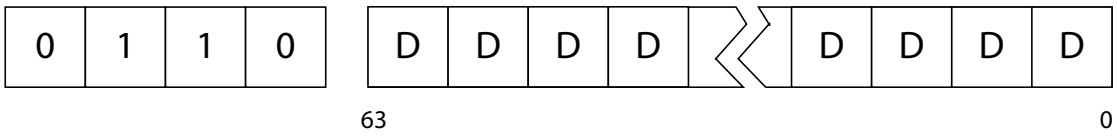

Figure A.7: Double and Tag

This tag indicates that the associated sixty-four D bits are one double-length IEEE 754 floating point number. The double is signed.

A.8 Reserved

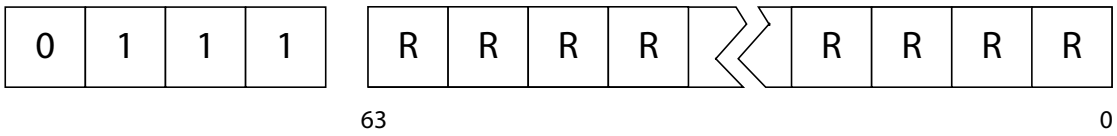

Figure A.8: Reserved

This tag is reserved for future use for an additional data type. Memory locations should not be tagged using this value.

A.9 Standard Object

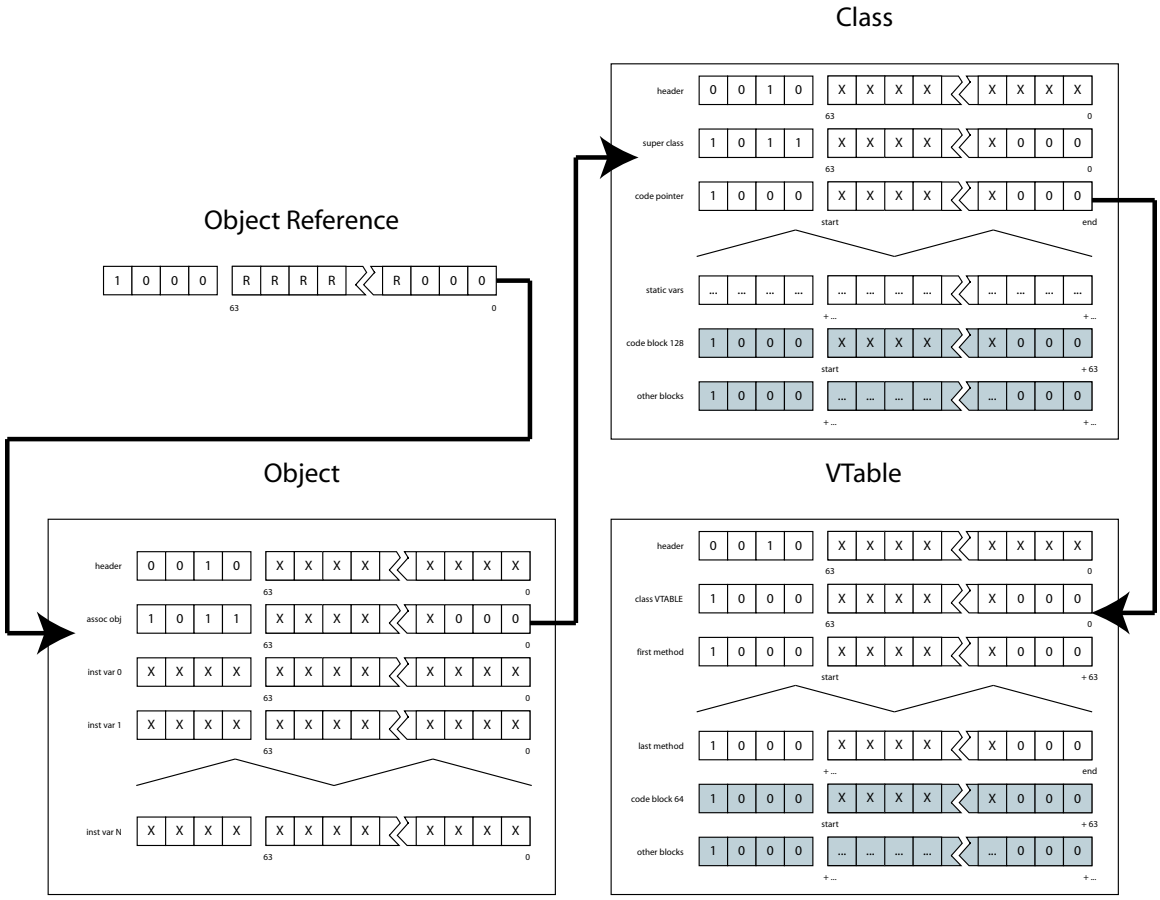

Figure A.9: A Standard Object Reference

This tag indicates that the associated sixty-four R bits are a reference to an object. The associated object reference of this object points to an associated class. The object does not have a code pointer.

Since memory in the virtual machine is byte addressable, and objects in the virtual machine must begin at sixty-four bit boundaries, the three least significant bits of object references are reserved for use by the garbage collector.

A.10 Object with Code

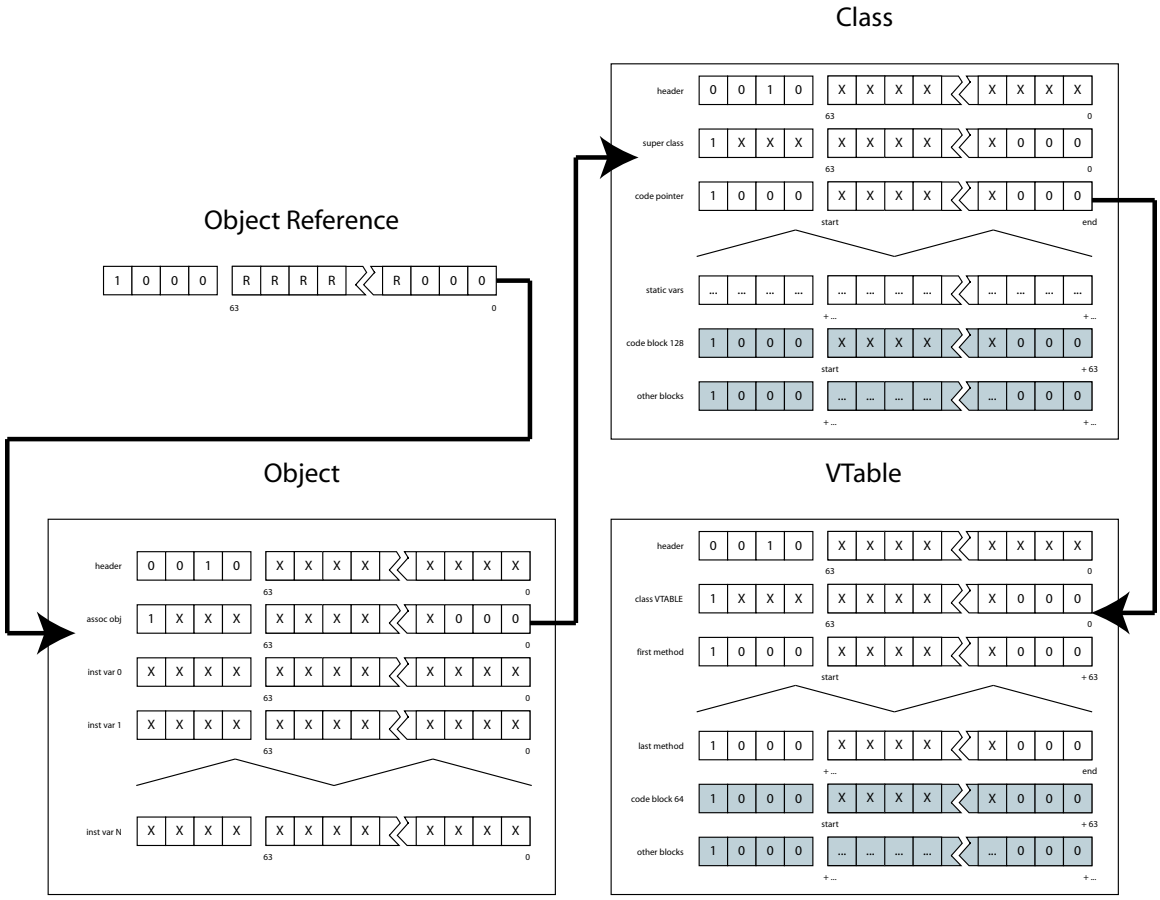

Figure A.10: Object with Code Reference

This tag indicates that the associated sixty-four R bits are a reference to an object. The associated object reference of this object points to an associated class. The object has a code pointer that points to a vtable of methods associated with it.

The three least significant bits of object references are reserved for use by the garbage collector.

A.11 List Element

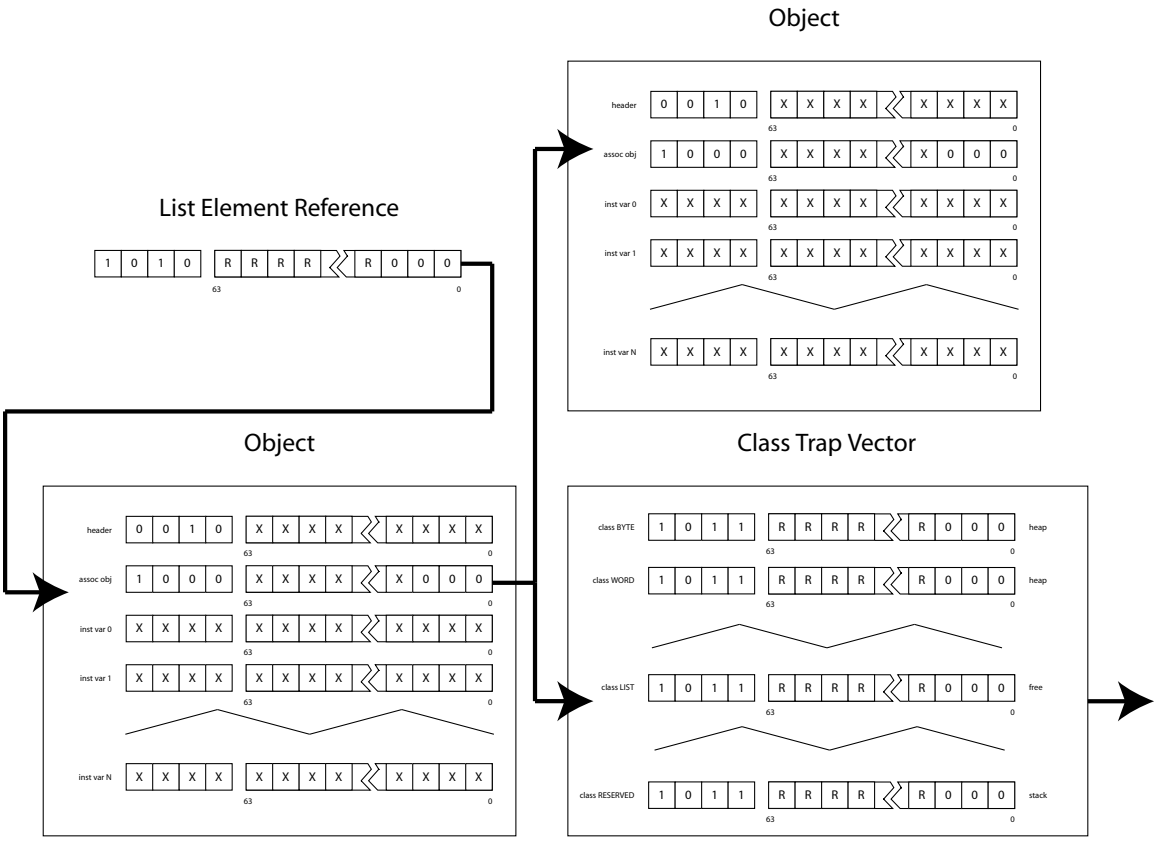

Figure A.11: List Element Reference

This tag indicates that the associated sixty-four R bits are a reference to a list element. The associated object reference of this object points to another object. The class associated with a list element can be determined via a virtual machine trap during code execution.

The three least significant bits of object references are reserved for use by the garbage collector.

A.12 Class Object

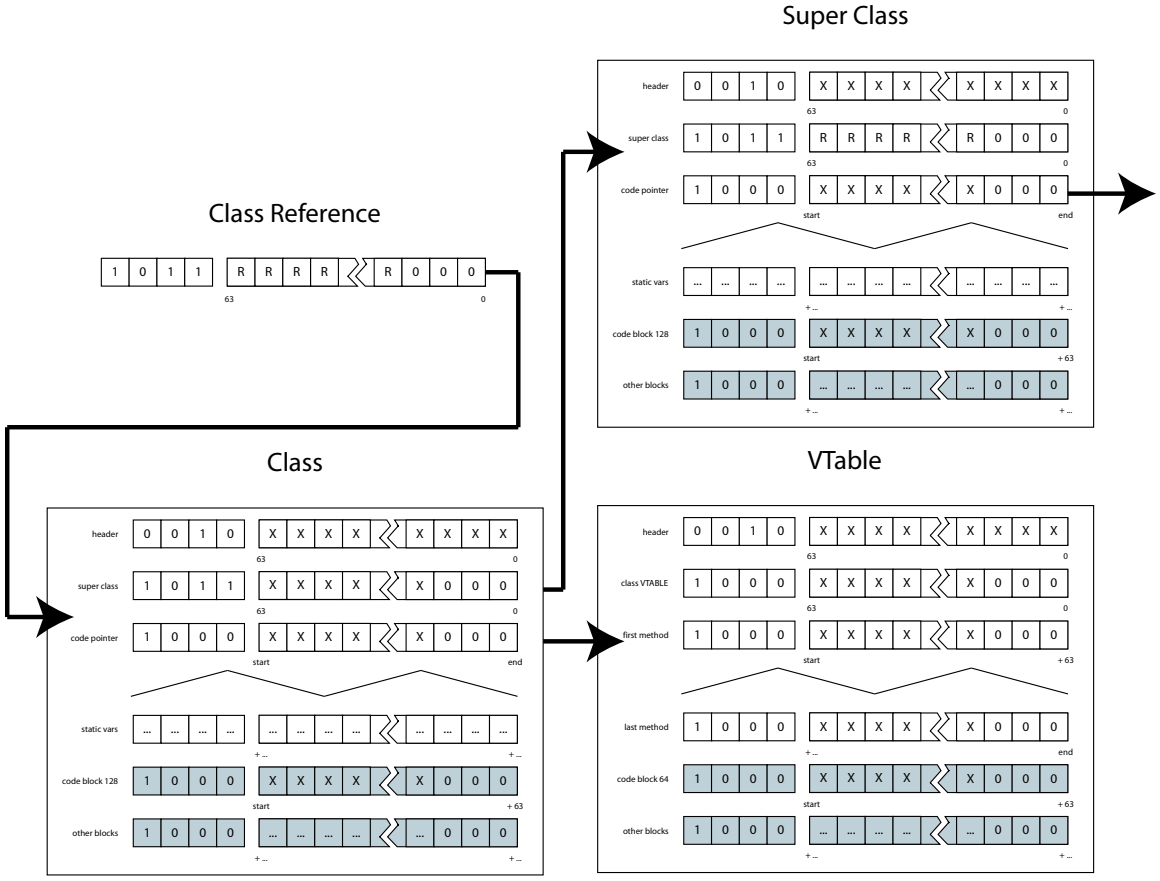

Figure A.12: Class Object Reference

This tag indicates that the associated sixty-four R bits are a reference to an object. The associated object reference of this class points to its superclass. The class has a code pointer that points to a vtable of methods associated with this class.

The three least significant bits of object references are reserved for use by the garbage collector.

A.13 Future Object

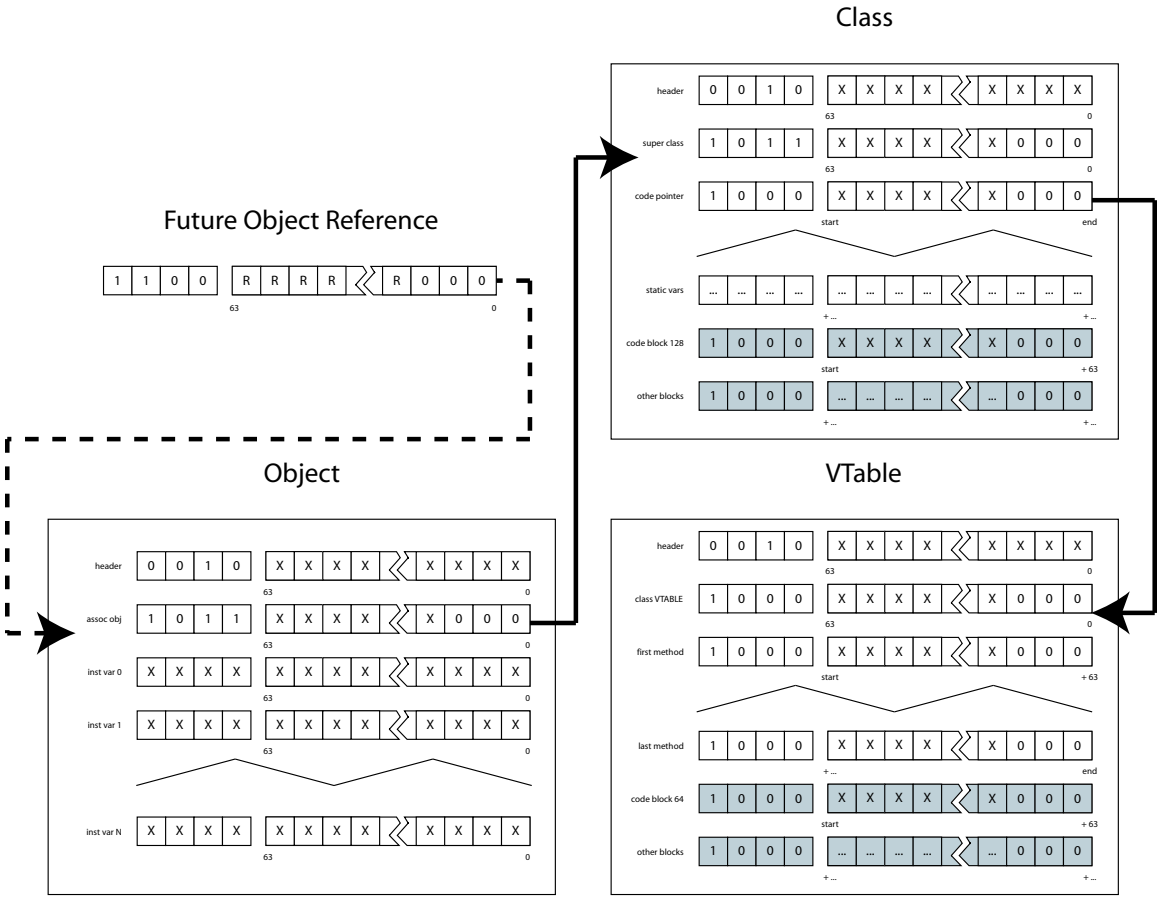

Figure A.13: Future Object Reference

This tag indicates that the associated sixty-four R bits are a reference to a future object, an object whose value has not yet been computed. The associated object reference of this object points to an associated class and is valid. This object does not have a code pointer.

The three least significant bits of object references are reserved for use by the garbage collector.

A.14 Future Object with Code

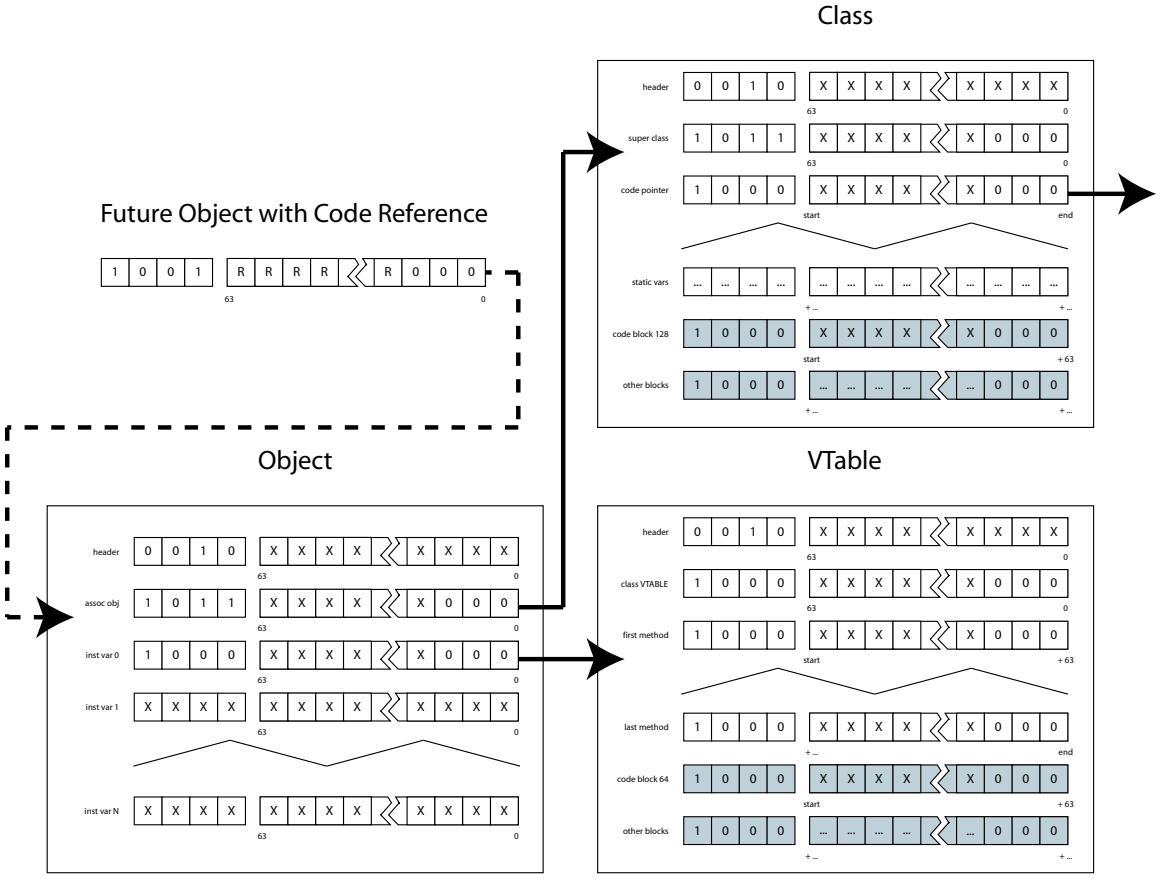

Figure A.14: Future with Code Reference

This tag indicates that the associated sixty-four R bits are a reference to an object, an object whose value has not yet been computed. The associated object reference of this object points to an associated class. The object has a code pointer that points to a vtable of methods associated with it. Both references are valid.

The three least significant bits of object references are reserved for use by the garbage collector.

A.15 Pointer

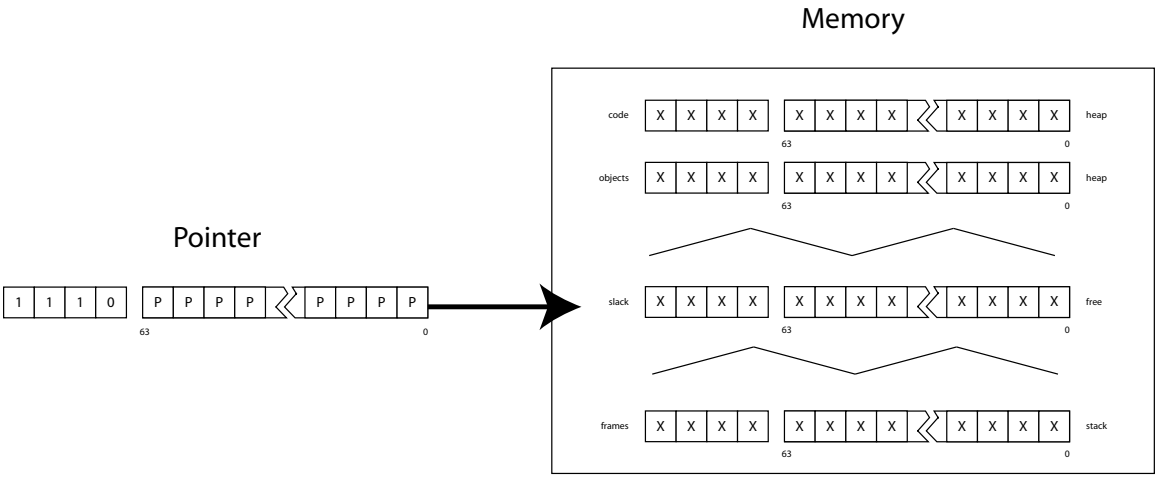

Figure A.15: Pointer Reference

This tag indicates that the associated sixty-four P bits are a reference to arbitrary data. This pointer need not point to the beginning of an object, and may point to memory with an arbitrary tag.

Since a pointer can point to byte-addressable memory, and since pointers do not directly participate in garbage collection, there are no free bits in the pointer type. All sixty-four bits are significant.

A.16 Reserved

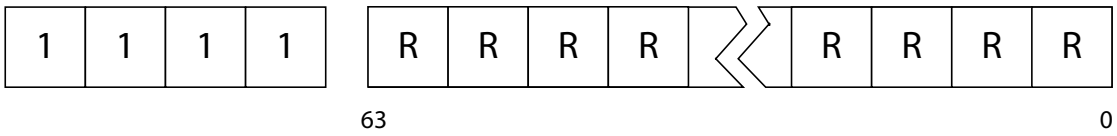

Figure A.16: Reserved

This tag is reserved for future use for an additional reference type. Memory locations should not be tagged using this value.

Appendix B

Opcodes

This appendix section lists the internal details of the 45 pre-defined opcodes available to the ++VM. This section gives a bit-by-bit description of each opcode, specifying the kinds of operations performed, the number of words necessary to represent these instructions, and the types of informational attributes available to each opcode. For a more general description of opcode behavior, see Chapter 8.

How To Read This Section

Title The opcode title. This also contains the three-letter shorthand abbreviation for the opcode.

Figure A picture of how the opcode is laid out in memory. The upper byte of word zero specifies the opcode number. All other white boxes specify required attributes, and gray boxes specify required attributes.

Description A brief summary of the operation that the opcode performs.

Required Attributes A list of attributes which all virtual machine implementations must recognize.

Informational Attributes A list of attributes which can be ignored by some virtual machine implementations but which can provide additional runtime information about the opcode.

Extension Words The number of additional instruction words needed by the opcode.

Exceptions A list of exceptions which may be thrown during the execution of this opcode.

Programming Notes Information about how to use this opcode.

B.1 ILLEGAL INSTRUCTION [ILL]

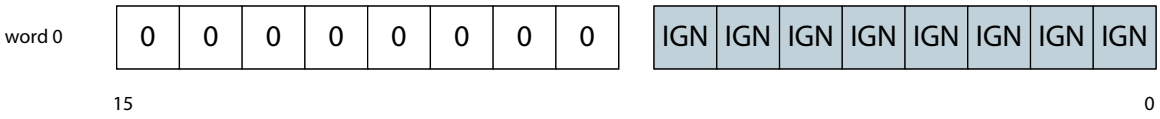

Figure B.1: Illegal Instruction

Description

This opcode is the illegal instruction, and should be executed if the virtual machine enters an inconsistent state. When executed, it will cause the virtual machine to throw an IllegalInstructionError. As an error, it is uncatchable and will cause the virtual machine to exit.

Attributes

The attribute byte of this instruction is ignored.

IGN The IGN bit is ignored.

Exceptions

Execution of this instruction can result in the following exceptions:

IllegalInstructionError Always thrown

Programming Notes

Programmers will be unlikely to explicitly use the illegal instruction in their code. It is put in place in case the virtual machine fails to correctly interpret an opcode and gets out of synch with the intended code or executes small data.

B.2 STACK OPERATIONS [STK]

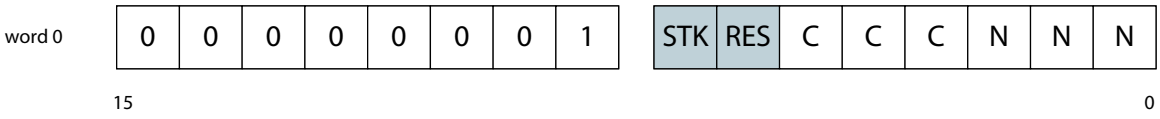

Figure B.2: Stack Operations

Description

This opcode performs one of eight stack operations. As of this writing, only four of the eight operations are defined: element duplication, element popping, element swapping, and element reversal. Stack instructions manipulate values on the top of the stack and do not depend upon tag information.

DUP The DUP instruction, NNN = 000, performs stack element duplication. The count amount specifies the number of duplication operations to perform.

POP The POP instruction, NNN = 001, performs stack popping. The count amount specifies the number of operations to perform.

SWP The SWP instruction, NNN = 010, performs stack element swapping. The count amount specifies the stack pointer offset of the element with which the top of the stack should be swapped.

REV The REV instruction, NNN = 011, performs stack element reversal. The count amount specifies the stack pointer offset of the element that will become the head of the stack; all items above it on the stack are popped off and placed back on the stack in reverse order.

RES The RES instruction, NNN = 1XX, is reserved for future extensions to the virtual machine.

Required Attributes

This instruction has two required attributes. The NNN attributes on the first instruction word specify the type of operation to perform. The CCC bits on the first instruction word specify the number of times to perform the operation, less one.

If NN is 000, then the virtual machine performs a DUP. If NN is 001, then a POP is performed. If NN is 001, then a swap is performed. If NN is 011, then a REV is performed. The remaining values of NNN instructions are reserved for future extensions to the virtual machine and should not be used.

The CCC bits specify the number of times the instruction should be repeated or the number of elements to be used in the operation, depending upon the context.

Informational Attributes

Two bits are available for informational attributes.

STK The STK bit on the first instruction word contains information about whether the stack is likely to be manipulated frequently. If STK is clear, then no information is known about stack manipulation and the virtual machine should assume that the stack will not be manipulated very often. If STK is set, then the stack will be manipulated frequently.

RES The remaining bit on the first instruction word is reserved for future extensions.

Extension Words

None, though future extensions may set the extension bit and use an additional extension word for attributes.

Exceptions

Execution of this instruction can result in the following exceptions.

EmptyStackException If insufficient stack values are available

B.3 ADDITION [ADD]

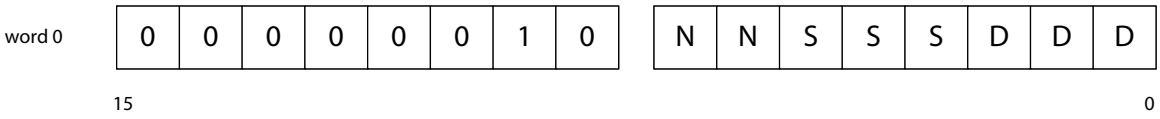

Figure B.3: Addition Instruction

Description

This opcode adds two data values. This operation is only defined on data registers. The source data register is added to the destination data register and the resulting value is stored in the destination data register. The size of the operation is determined by the tag on the source and destination.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

B.4 SUBTRACTION [SUB]

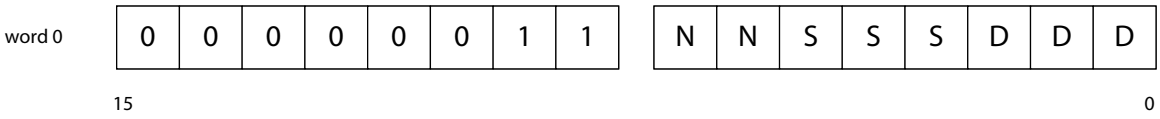

Figure B.4: Subtraction Instruction

Description

This opcode subtracts one data value from another. This operation is only defined on data registers. The source data register is subtracted from the destination register and the resulting value is stored in the destination data register. The size of the operation is determined by the tag on the source and destination.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

B.5 MULTIPLICATION [MUL]

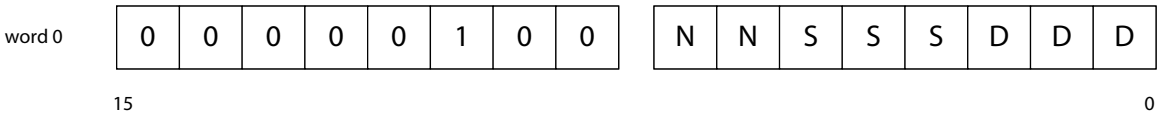

Figure B.5: Multiplication Instruction

Description

This opcode adds multiplies two data values. This operation is only defined on data registers. The source data register is multiplied with the destination register and the resulting value is stored in the destination data register. The size of the operation is determined by the tag on the source and destination.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

B.6 DIVISION [DIV]

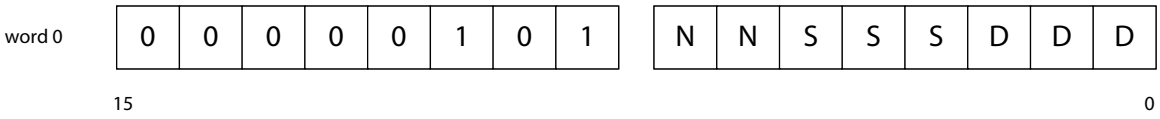

Figure B.6: Division Instruction

Description

This opcode divides one data value by another. This operation is only defined on data registers. The destination data register is divided by the source register and the resulting value is stored in the destination data register. The size of the operation is determined by the tag on the source and destination.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

DivideByZeroException If the divisor is zero

B.7 MODULUS [MOD]

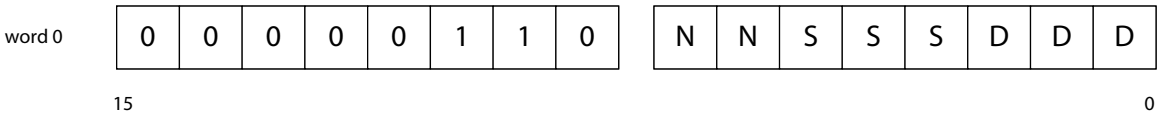

Figure B.7: Modular Arithmetic Instruction

Description

This opcode performs modular arithmetic by dividing one data value by another and keeping the remainder. This operation is only defined on data registers. The destination data register is divided by the source data register and the remainder from the division is stored back in the destination data register. The tag of the resulting value will be determined by the tag on the source and destination registers.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

DivideByZeroException If the divisor is zero

B.8 NUMERIC INNER PRODUCT [NPR]

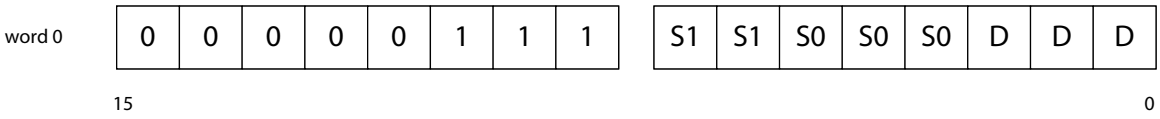

Figure B.8: Numeric Inner Product Instruction

Description

This opcode multiplies two numbers and adds the resulting value to a third. The numeric inner product is only defined on data registers. The two source operands are multiplied together, and the subsequent result is is added to the destination register. The size of the operation performed is determined by the tag of the source and destination; in order for the operation to succeed, both of the sources and the destination must have the same tag.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The source S0 specifies any one of the eight data registers, while the source S1 specifies one of the four low data registers (%d0 to %d3).

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags of all three operands are not equal

B.9 LOGICAL SHIFT RIGHT [LSR]

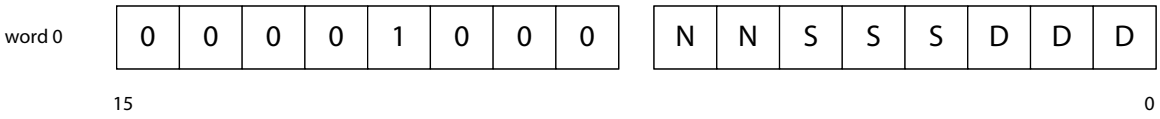

Figure B.9: Logical Shift Right Instruction

Description

This opcode performs a logical shift right operation. This operation is only defined on data registers. The destination data register is shifted right by the number of bits specified in the source data register and the resulting value is stored back in the destination data register. The bits inserted to the left of the shifted number are zeros. The size of the operation will be determined by the tag of the destination register.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

ShiftException If the shift amount is negative or malformed.

B.10 ARITHMETIC SHIFT RIGHT [LSR]

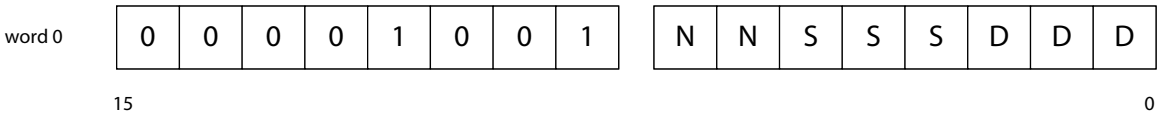

Figure B.10: Arithmetic Shift Right Instruction

Description

This opcode performs an arithmetic shift right operation. This operation is only defined on data registers. The destination data register is shifted right by the number of bits specified in the source data register and the resulting value is stored back in the destination data register. The bits inserted to the left of the shifted number have the same bit value as the sign bit of the original operand. The size of the operation will be determined by the tag of the destination register.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

ShiftException If the shift amount is negative or malformed.

Programming Notes

The Arithmetic Shift Left [ASL] operation is a synonym for the Logical Shift Left operation.

B.11 LOGICAL SHIFT LEFT [LSL]

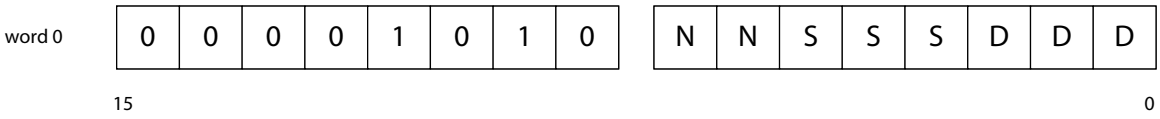

Figure B.11: Logical Shift Left Instruction

Description

This opcode performs a logical shift left operation. This operation is only defined on data registers. The destination data register is shifted left by the number of bits specified in the source data register and the resulting value is stored back in the destination data register. The bits inserted to the right of the shifted number are zeros. The size of the operation will be determined by the tag of the destination register.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

ShiftException If the shift amount is negative or malformed.

B.12 BIT OPERATIONS [BOP]

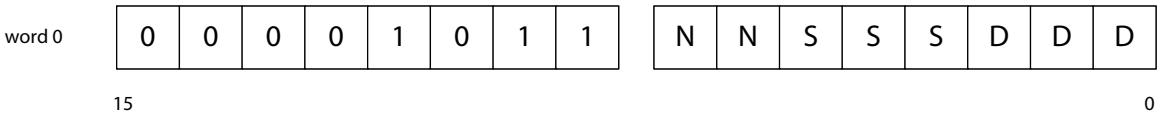

Figure B.12: Bit Operation Instruction

Description

This opcode performs various manipulations to bit fields. Bit field manipulation operations are only defined for data registers. The tag of the resulting value will be determined by the tag on the source and destination registers.

CLR The CLR instruction, NN = 00, performs a bit clearing operation. The bits specified by the source register are set to zero in the destination register.

SET The SET instruction, NN = 01, performs a bit setting operation. The bits specified by the source register are set to one in the destination register.

TGL The TGL instruction, NN = 10, performs a bit toggling operation. The bits specified by the source register are toggled in the destination register; if the bit was a zero it is toggled to a one, and if a one it is toggled to a zero.

RES The RES instruction, NN = 11, is reserved for future extensions to the virtual machine.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, then the bits that SSS specifies should be cleared. If NN is 01, the bits that SSS specifies should be set. If NN is 10, the bits that SSS specifies should be toggled. The NN value of 11 is reserved and should not be used.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags of both operands are not equal

B.13 SET UNION [UNN]

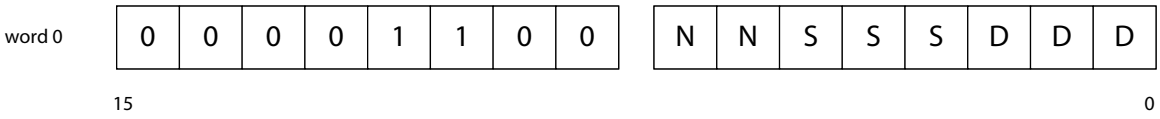

Figure B.13: Set Union Instruction

Description

This opcode performs a set union operation by taking the bitwise OR of two data values. This operation is only defined on data registers. The destination data register is ORed with the source data register and the resulting value is stored back in the destination data register. The tag of the resulting value will be determined by the tag on the source and destination registers.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

B.14 SET INTERSECTION [NTR]

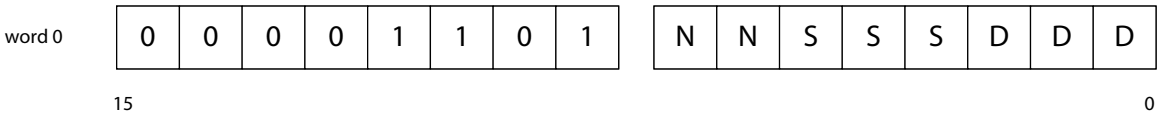

Figure B.14: Set Intersection Instruction

Description

This opcode performs a set intersection operation by taking the bitwise AND of two data values. This operation is only defined on data registers. The destination data register is ANDed to the source data register and the resulting value is stored back in the destination data register. The tag of the resulting value will be determined by the tag on the source and destination registers.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

B.15 SET EXCLUSIVE OR [XOR]

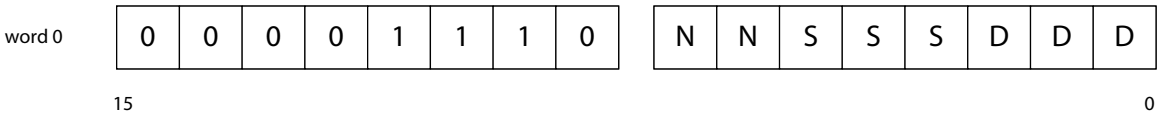

Figure B.15: Set Exclusive Or Instruction

Description

This opcode performs a set exclusive or operation by taking the bitwise EXCLUSIVE OR of two data values. This operation is only defined on data registers. The destination data register is EXCLUSIVE ORed with the source data register and the resulting value is stored back in the destination data register. The tag of the resulting value will be determined by the tag on the source and destination registers.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, the three SSS bits indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

B.16 SET INNER PRODUCT [SPR]

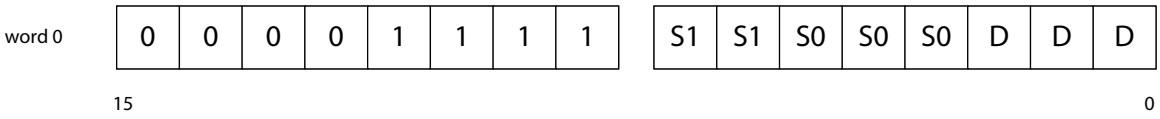

Figure B.16: Set Inner Product Instruction

Description

This opcode performs a set inner product instruction, ANDing two operands together and ORing that result with a third value. The set inner product is only defined on data registers. A bitwise AND of the two source operands is performed, then a bitwise OR is performed with the intermediate result and a destination register. The size of the operation performed is determined by the tag of the source and destination.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination register. The source S0 specifies any one of the eight data registers, while the source S1 specifies one of the four low data registers (%d0 to %d3).

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags of all three operands are not equal

B.17 CONVERT [CON]

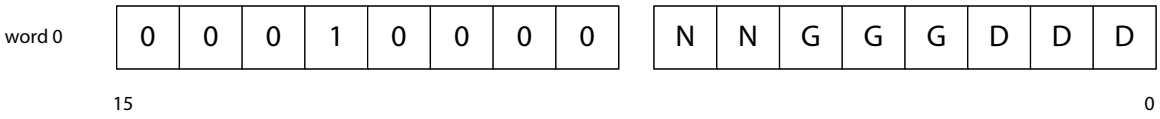

Figure B.17: Convert Instruction

Description

This opcode retags a value to the specified data type. The source and destination registers must have the same register number, though it is possible for either the source or destination to be a value in a reference register. Some conversions may lose precision.

Required Attributes

All bits in the attribute word are required. The DDD bits of the first instruction word specify the (source and) destination register. The GGG bits specify the tag to which the source should be converted. The NN bits specify whether the registers should be data or reference registers.

If NN is 00, both operands should come from data registers. If NN is 01, the source is a data register and the destination is a reference register. If NN is 10, the source is a reference register and the destination is a data register. If NN is 11, the source and destination registers are both reference registers.

Execution of this instruction will not result in exceptions.

B.18 CAST [CST]

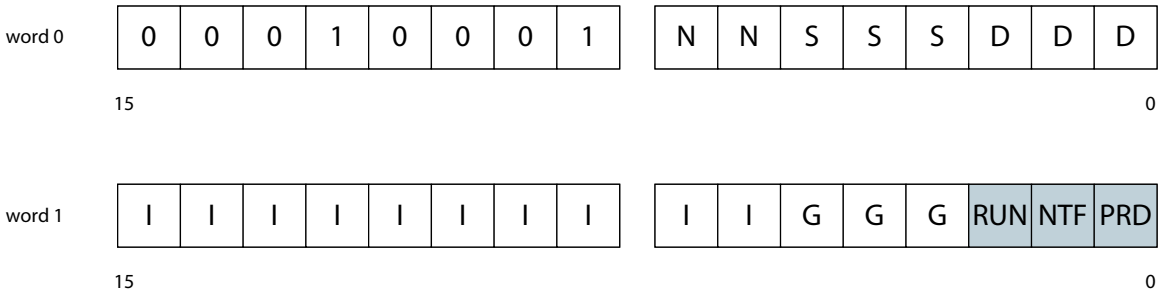

Figure B.18: Cast Instruction

Description

This opcode casts a value to an instance of a specified class. The destination register is a reference register; the source register is either a data register or a reference register depending upon an attribute bit. The result is tagged with the same tag type as the source if the source is an reference; the result is tagged as a standard object if the source is a data value.

Required Attributes

All bits in the attribute word are required attributes. The DDD bits specify the destination register. The SSS bits specify the source register. The NN bits specify the type of reference to create.

Thirteen bits on the next extension word are used for required attributes. Ten bits of the second instruction word specify an index in the constant pool, the index of the class to which the value should be cast. The GGG bits specify the reference type to which the object should be converted.

If NN is 00, both operands should come from data registers. If NN is 01, the source is a data register and the destination is a reference register. If NN is 10, the source is a reference register and the destination is a data register. If is 11, the source and destination registers are both reference registers.

If GGG is 000, the value should be converted to an object. If GGG is 001, the value should be converted to an object with code. If GGG is 010, the value should be converted to a list element. If GGG is 011, the value should be converted to a

class. If GGG is 100, the value should be converted to a future object. If GGG is 101, the value should be converted to a future object with code. If GGG is 110, the value should be converted to a pointer. The GGG value of 111 is reserved for future extensions to the type system

Informational Attributes

Three bits are available for informational attributes.

RUN The RUN bit contains information about whether a runtime check will be necessary to determine that this cast must be performed. If RUN is clear, the virtual machine defaults and the runtime test is performed. If RUN is set, however, it will be assumed that the cast always succeeds and no runtime tests will be performed.

NTF The NTF bits contain information about whether to check for implemented interfaces before implemented classes when determining whether the cast succeeds. If NTF is clear, then superclasses are checked before interfaces. If NTF is set, then implemented interfaces are checked before superclasses.

PRD The PRD bit contains prediction information about the cast. If PRD is clear, then the cast is likely to succeed and the instruction steam should contain the next opcode. If PRD is set, then the cast is likely to fail and the instruction stream should prepare to throw an exception.

Extension Words

One extension word holds the opcode information.

Exceptions

Execution of this instruction can result in the following exceptions

ClassCastException A ClassCastException is thrown if the cast fails

Programming Notes

This instruction is dangerous and potentially difficult to use.

B.19 ONE OPERAND INSTRUCTIONS [OOP]

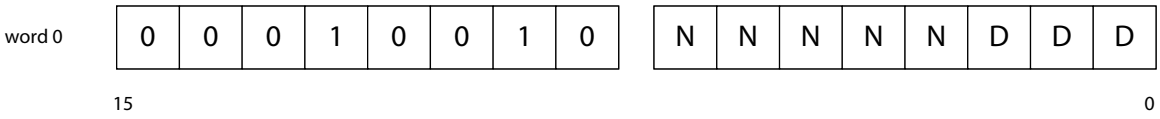

Figure B.19: One Operand Instruction

Description

This opcode executes one of various one-operation instructions. These operations are all alike in the fact that they all use only a single operand, meaning that no source register need be specified. Currently, seven one-op instructions are defined.

NOT The NOT instruction, NNNNN = 00000, performs one's complement negation. The destination register is a data register.

NEG The NEG instruction, NNNNN = 00001, performs two's complement negation. The destination register is a data register.

ILB The ILB instruction, NNNNN = 00010, computes the index of the leftmost set bit. This value is approximately equal to the number taken log base 2. The destination register is a data register.

IRB The IRB instruction, NNNNN = 00011, computes the index of the rightmost set bit. The destination register is a data register.

SZE The SZE instruction, NNNNN = 00101, returns the size of an object from that object's header field. The destination is a reference register that holds an object; the register will be re-tagged as a short as a result of computing this instruction

CNT The CNT instruction, NNNNN = 00110, computes the number of bits that are set. This value is an approximation count. The destination register is a data register.

RES The RES instructions, all other values of NNNNN, are reserved for future extensions to the virtual machine.

Required Attributes

All of the bits in the attribute byte are used. The DDD bits specify the number of a (source and) destination register. The NNNNN bits specify the operation to perform.

If NNNNN is 000, perform a NOT operation, as defined above. If NNNNN is 001, perform a NEG operation. If NNNNN is 010, perform an ILB operation. If NNNNN is 011, perform an IRB operation. If NNNNN is 100, perform a FLD operation. If NNNNN is 101, perform a TRP operation. If NNNNN is CNT, perform a CNT operation. All other values of NNNNN are reserved and should not be used

B.20 MOVE BETWEEN REGISTERS [MRG]

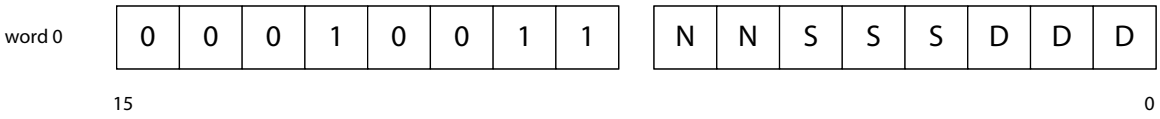

Figure B.20: Move Between Register Instruction

Description

This opcode moves a value between two registers or between a register and the stack. The source value is moved to the destination register as appropriate. Movement operations can occur between two data registers, between two reference registers, or from a data register to a reference register. When values are moved between two data or two reference registers, the data tag is preserved; when an reference is converted to a data value, any reference is silently converted to a long.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits specify the number of the destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, both the source and the destination are data registers and SSS is a data register. If NN is 01, the source is a special value as explained below. If NN is 10, the source is a reference register and the the destination is a data register and SSS is a reference register. If NN is 11, the source and destination are both reference registers and SSS is an reference register.

If NN is 01, then the SSS bits do not specify a register. Instead, if SSS is 000, then the top of the stack is the source and a the destination is a data register. If SSS is 001, then the top of the stack is the source and the destination is a reference register. If SSS is 010, then source is an ultra from the stack and the destination is the lower of two data registers. If SSS is 011, then one extension word holds an index into a constant pool and the destination is a data register. If SSS is 100, the source is the data register specified by the DDD field and the destination is the stack. If SSS is 101, the source is the reference register specified by the DDD field and the destination is the stack. If SSS is 110, then the source is the ultra specified in register

DDD and the source is the stack. If SSS is 111, one extension word holds an index into the constant pool and the destination is a reference register.

Extension Words

One extension word may be needed to hold an index into the constant pool.

Programming Notes

This instruction can be used to access constants in the constant pool and store them into registers.

B.21 LOAD FROM OBJECT OFFSET [LOB]

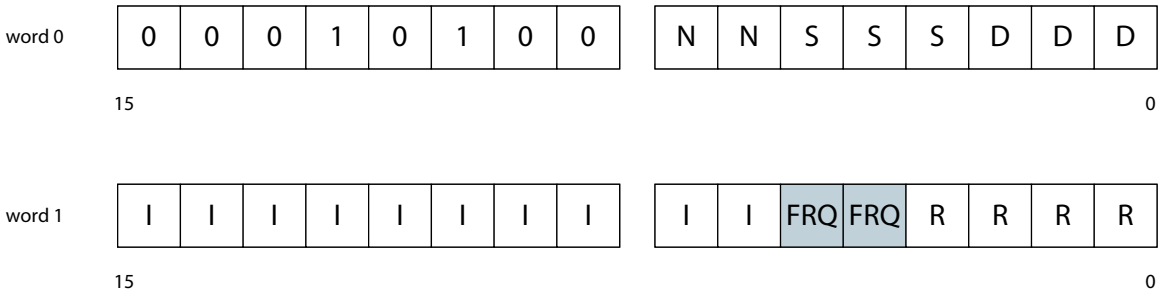

Figure B.21: Load from Object Instruction

Description

This opcode moves a value from a specified object offset into the specified register. The memory offset is computed as an offset from a base object. The tag of the specified memory location will determine whether a data or reference register is used as the destination.

Required Attributes

All of the bits in the attribute byte are used. The DDD bits specify the number of a destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, then the base register is the %to register and the offset SSS is an eight-bit offset index. If NN is 01, then the base register is the %tc register and the offset SSS is an eight-bit offset index. If NN is 10, then the base register is the current frame and the offset SSS is an eight-bit offset index. If NN is 11, then an additional extension word is needed to handle a special case and SSS is ignored.

If NN is 11, the first ten bits of the next extension word form an index to the specified object field and the RRRR bits of the word specify the location of the base register. If the last RRRR bits are of the pattern 0RRR, the base register is the reference register RRR. The pattern 1000 specifies the %to reference, 1001 specifies the %tc reference, and all other patterns are reserved for future extensions to the virtual machine.

Informational Attributes

If the instruction uses a single word, then there are no informational attributes; if an additional word is used, however, there are two informational bits available.

FRQ The two FRQ bits contain information about how frequently the memory location will be accessed in the future; the system may be able to use this information to cache more efficiently. If FRQ is 00, then no access information information is known. If FRQ is 01, then the object will not be accessed frequently. If FRQ is 10, then the object will be accessed relatively frequently in the near future. If FRQ is 11, then the object should be kept as long as possible in the cache.

Extension Words

One additional extension word can be used to hold the base offset register as well as additional attribute information.

B.22 STORE TO OBJECT OFFSET [SOB]

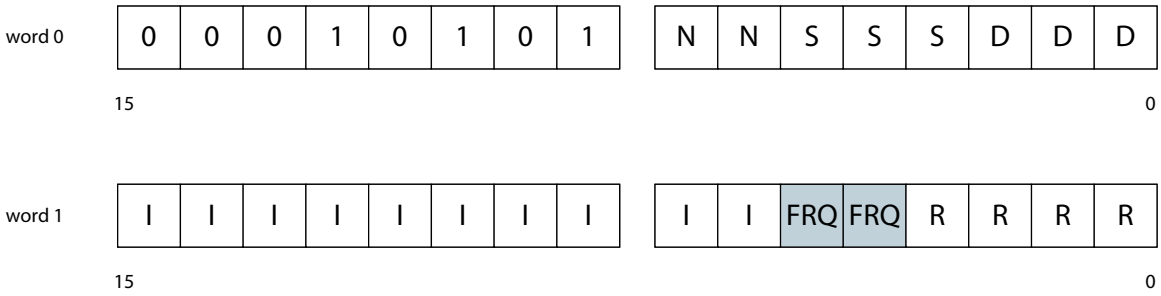

Figure B.22: Store to Object Instruction

Description

This opcode moves a value from a source register to a specified object offset. The memory offset is computed as an offset from a base object. The tag of the specified memory location will determine whether a data or reference register is used as the destination.

Required Attributes

All of the bits in the attribute byte are used. The DDD bits specify the number of a source register. The five remaining bits on the first instruction word specify the opcode destination.

If NN is 00, then the base register is the %to register and the offset SSS is an eight-bit offset index. If NN is 01, then the base register is the %tc register and the offset SSS is an eight-bit offset index. If NN is 10, then the base register is the current frame and the offset SSS is an eight-bit offset index. If NN is 11, then an additional extension word is needed to handle a special case and SSS is ignored.

If NN is 11, the first ten bits of the next extension word form an index to the specified object field and the RRRR bits of the word specify the location of the base register. If the last RRRR bits are of the pattern 0RRR, the base register is the reference register RRR. The pattern 1000 specifies the %to reference, 1001 specifies the %tc reference, and all other patterns are reserved for future extensions to the virtual machine.

Informational Attributes

If the instruction uses a single word, then there are no informational attributes; if an additional word is used, however, there are two informational bits available.

FRQ The two FRQ bits contain information about how frequently the memory location will be accessed in the future; the system may be able to use this information to cache more efficiently. If FRQ is 00, then no access information information is known. If FRQ is 01, then the object will not be accessed frequently. If FRQ is 10, then the object will be accessed relatively frequently in the near future. If FRQ is 11, then the object should be kept as long as possible in the cache.

Extension Words

One additional extension word can be used to hold the base offset register as well as additional attribute information.

B.23 LOAD FROM POINTER OFFSET [LPT]

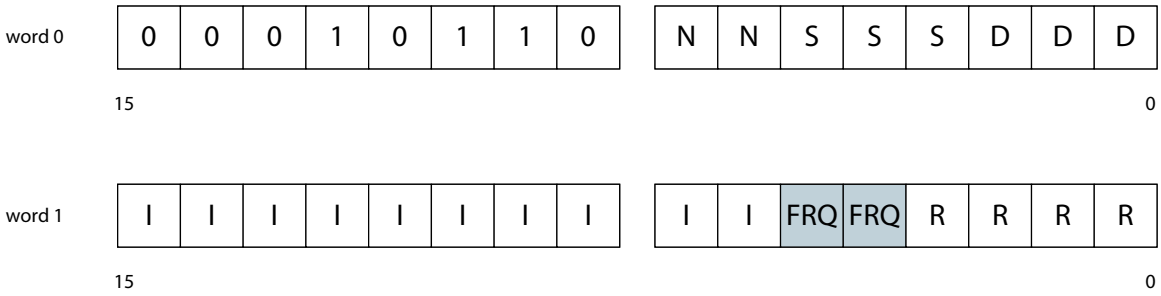

Figure B.23: Load from Pointer Instruction

Description

This opcode moves a value from a specified object offset into the specified register. The memory offset, stored in a register, is computed as an offset from a base pointer. The tag of the specified memory location will determine whether a data or reference register is used as the destination.

Required Attributes

All of the bits in the attribute byte are used. The DDD bits specify the number of a destination register. The five remaining bits on the first instruction word specify the opcode source.

If NN is 00, then the base register is the %to register and the offset SSS is a data register. If NN is 01, then the base register is the %tc register and the offset SSS is a data register. If NN is 10, then the base register is the current frame and the offset SSS is a data register. If NN is 11, then an additional extension word is needed to handle a special case and SSS is ignored.

If NN is 11, the first ten bits of the next extension word form an index to the specified object field and the RRRR bits of the word specify the location of the base register. If the last RRRR bits are of the pattern 0RRR, the base register is the reference register RRR. The pattern 1000 specifies the %to reference, 1001 specifies the %tc reference, and all other patterns are reserved for future extensions to the virtual machine.

Informational Attributes

If the instruction uses a single word, then there are no informational attributes; if an additional word is used, however, there are two informational bits available.

FRQ The two FRQ bits contain information about how frequently the memory location will be accessed in the future; the system may be able to use this information to cache more efficiently. If FRQ is 00, then no access information information is known. If FRQ is 01, then the object will not be accessed frequently. If FRQ is 10, then the object will be accessed relatively frequently in the near future. If FRQ is 11, then the object should be kept as long as possible in the cache.

Extension Words

One additional extension word can be used to hold the base offset register as well as additional attribute information.

B.24 STORE TO POINTER OFFSET [SPT]

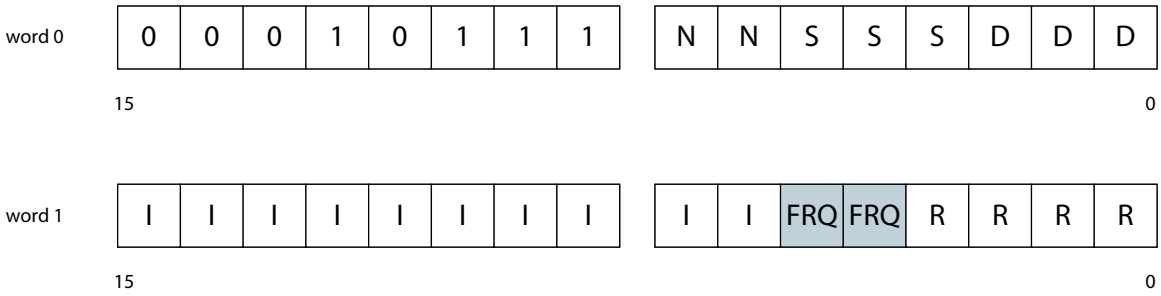

Figure B.24: Store to Pointer Instruction

Description

This opcode moves a value from a source register to a specified object offset. The memory offset, stored in a pointer, is computed as an offset from a base pointer. The tag of the specified memory location will determine whether a data or reference register is used as the destination.

Required Attributes

All of the bits in the attribute byte are used. The DDD bits specify the number of a source register. The five remaining bits on the first instruction word specify the opcode destination.

If NN is 00, then the base register is the %to register and the offset SSS is a data register. If NN is 01, then the base register is the %tc register and the offset SSS is a data register. If NN is 10, then the base register is the frame pointer and the offset SSS is a data register. If NN is 11, then an additional extension word is needed to handle a special case and SSS is ignored.

If NN is 11, the first ten bits of the next extension word form an index to the specified object field and the RRRR bits of the word specify the location of the base register. If the last RRRR bits are of the pattern 0RRR, the base register is the reference register RRR. The pattern 1000 specifies the %to reference, 1001 specifies the %tc reference, and all other patterns are reserved for future extensions to the virtual machine.

Informational Attributes

If the instruction uses a single word, then there are no informational attributes; if an additional word is used, however, there are two informational bits available.

FRQ The two FRQ bits contain information about how frequently the memory location will be accessed in the future; the system may be able to use this information to cache more efficiently. If FRQ is 00, then no access information information is known. If FRQ is 01, then the object will not be accessed frequently. If FRQ is 10, then the object will be accessed relatively frequently in the near future. If FRQ is 11, then the object should be kept as long as possible in the cache.

Extension Words

One additional extension word can be used to hold the base offset register as well as additional attribute information.

B.25 PREPARE NEW OBJECT [NEW]

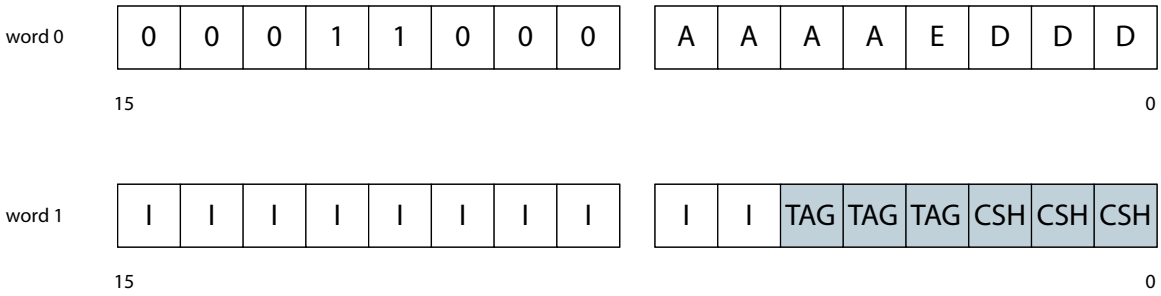

Figure B.25: Prepare New Object Instruction

Description

This opcode allocates space for a new object reference. Memory is prepared for an instance of the class held at the specified constant pool index, and the memory reference to the new object is placed in the destination reference register.

Required Attributes

All of the bits in the attribute byte are used for required attributes. The DDD bits specify the destination reference register that will hold the pointer to the new object. The E bit specifies whether the instruction has an additional word's worth of informational attributes. The N bit specifies whether an object or an array is being created.

If N is 0, then a single object will be created. If N is 1, then the reference register DDD contains the length of the array to initialize.

Ten bits of the second instruction word specify an index in the constant pool, the index of the class to initialize. If an array is being created, this index is to the types of elements that will be in the array. If a single object is being created, this specifies the class of the object to create.

Informational Attributes

Six bits on the second extension word can be used to hold informational attributes.

PIN The two PIN bits contain information about object pinning; the system can use this information to prevent the garbage collector from moving a particular object so pointer operations can be performed. If PIN is 00, then no pinning information is available. If PIN is 01, then the object should be pinned in place. If PIN is 10, then the object should be pinned at the location specified by the next four extension words. If PIN is 11, then the object should be created in an unpinned fashion.

STK The STK bit contains information about whether the object shoud be allocated on the stack. If STK is 0, no information about stack allocation is avaliable. If STK is 1, then the object may be stack allocated.

TAG The three TAG bits indicate the tag that should be used when creating the object. This attribute can be used to optimize for the creation of normal objects, objects with code, futures, and pointers.

CSH The three CSH bits indicate another reference register with which the new object will be frequently used. The virtual machine should move the object in the destination register so that it does not create cache conflicts with the object in this register.

Extension Words

If the additional word attribute is set, one additional extension word can be used to hold more informational attributes. Four additional extension words may be needed if the object is pinned.

Programming Notes

This instruction allocates space for a new object, but does not initialize it. The resulting object should be initialized by calling the initializer method.

B.26 OBJECT ATTRIBUTE [ATR]

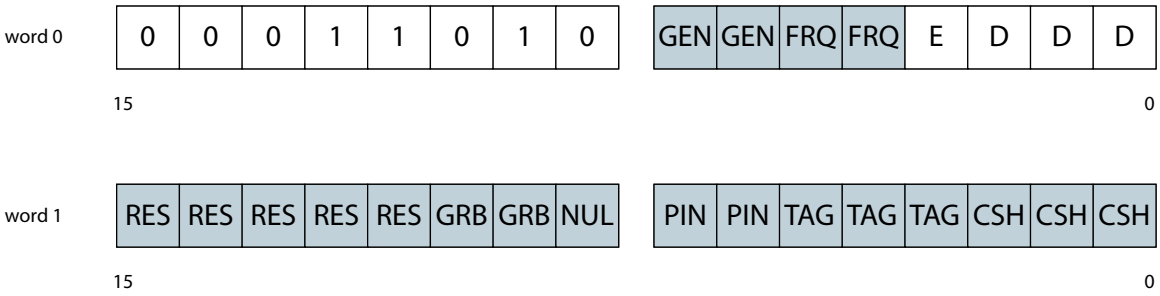

Figure B.26: Object Attribute Instruction

Description

This opcode provides additional runtime information about an object that has already been created. The object attribute instruction is only defined on values in a reference registers.

Required Attributes

Four bits in the attribute byte are used for required attributes. The DDD bits specify the destination reference register that will hold the pointer to the new object. The E bit specifies whether the instruction has an additional word's worth of informational attributes.

Informational Attributes

Twenty bits can be used to hold informational attributes.

GEN The two GEN bits contain generational information about the object; the system may be able to use this information to garbage collect more efficiently. If GEN is 00, then no information is known. If GEN is 01, the object will die and can be garbage collected very shortly in the future. If GEN is 10, the object will not die in the near future. If GEN is 11, then the object will live for a very long time and the garbage collector may not want to try to garbage collect it for a long period of time.

FRQ The two FRQ bits contain information about how frequently the object will be accessed; the system may be able to use this information to cache more efficiently. If FRQ is 00, then no access information information is known. If FRQ is FRQ, then the object will not be accessed frequently. If FRQ is 10, then the object will be accessed relatively frequently in the near future. If FRQ is 11, then the object should be kept as long as possible in the cache.

GRB The two GRB bits contain information about object garbage collection; the system can use this information to perform garbage collection more efficiently. If GRB is 00, then no garbage collection is available. If GRB is 01, then the garbage collector should be run immediately on the object because it contains many garbage references. If GRB is 10, then garbage collection should be disabled for it. If GRB is 11, then garbage collection should be re-enabled for the object.

NUL The NUL bit contains information about whether the object will ever be null. If NUL is clear, then the object may or may not be null and the system should make no assumptions about null checks. If NUL is set, the object will never be null and the null checks can be avoided.

PIN The two PIN bits contain information about object pinning; the system can use this information to prevent the garbage collector from moving a particular object so pointer operations can be performed. If PIN is 00, then no pinning information is available. If PIN is 01, then the object should be pinned in place. If PIN is 10, then the object should be pinned at the location specified by the next four extension words. If PIN is 11, then the object should be unpinned and can be moved by the garbage collector.

TAG The three TAG bits indicate the tag that should be used when creating the object. This attribute can be used to optimize for the creation of normal objects, objects with code, futures, and pointers.

CSH The three CSH bits indicate another reference register with which the object will be frequently used. The virtual machine should move the object in the destination register so that it does not create cache conflicts with the object in this register.

RES The remaining five bits are reserved for future extensions to the virtual machine.

Extension Words

If the additional word attribute is set, one additional extension word can be used to hold more informational attributes. Four additional extension words may be needed if the object is pinned.

Programming Notes

This opcode provides informational attributes to the runtime system.

B.27 CREATE A METHOD [MTH]

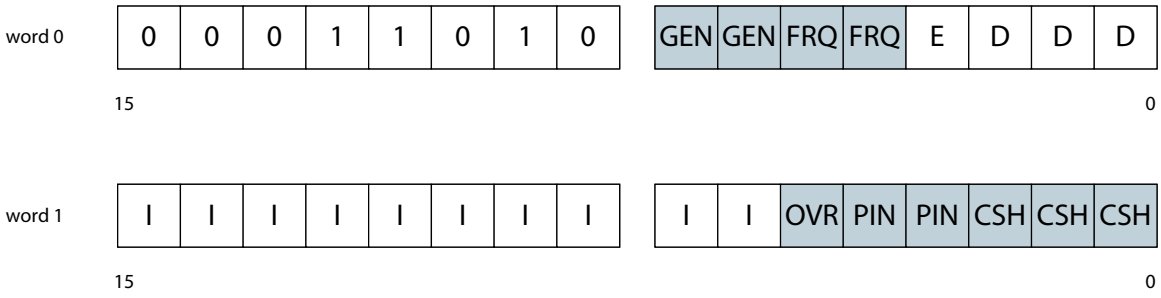

Figure B.27: Prepare Method Instruction

Description

This opcode creates a method from an array of words. The array of words is held in a destination register. Certain languages may require method verification when the method is created.

Required Attributes

All of the bits in the attribute byte are used for required attributes. The DDD bits specify the destination reference register that will hold the pointer to the new object. The E bit specifies whether the instruction has an additional word's worth of informational attributes.

Ten bits on the next extension word specify the constant pool index of the method to create or overwrite.

Informational Attributes

Ten bits can be used to hold informational attributes.

FRQ The two FRQ bits specify how frequently the method will be accessed and may provide information that is helpful to the cache. If FRQ is 00, then no access information is available. If FRQ is 01, then the method will be accessed infrequently. If FRQ is 10, then the method will be accessed frequently. If FRQ is 11, then the method will be accessed extremely frequently.

GEN The two GEN bits specify how long the method will be needed, and provides a hint to the virtual machine about possibly unloading the method in the future. If GEN is 00, then no generational information is provided. If GEN is 01, then the method will only be around for a short period of time. If GEN is 10, then the method will be around for a long period of time. If GEN is 11, then the method will never be unloaded from the virtual machine.

OVR The OVR bit specifies whether this method definition overwrites a current entry in the class's vtable. If OVR is clear, the method is added to the end of the vtable. If OVR is set, the method overwrites a current entry in the vtable. If OVR is clear but a method already exists at the vtable location, an exception should be thrown.

PIN The two PIN bits contain information about method pinning; the system can use this information to prevent the garbage collector from moving a particular method so pointer operations can be performed. If PIN is 00, then no pinning information is available. If PIN is 01, then the method should be pinned in place. If PIN is 10, then the method should be pinned at the location specified by the next four extension words. If PIN is 11, then the method should be unpinned and can be moved by the garbage collector.

CSH The three CSH bits indicate another reference register with which this method will be frequently called. The virtual machine should place the method in such a way so that it does not create cache conflicts with the method in this register.

Extension Words

One, but future implementations may use the extension bit to add additional information attributes.

Exceptions

Execution of this instruction can result in the following exceptions:

InvalidMethodException If the array of words is not a valid method

B.28 CREATE A CLASS [CLS]

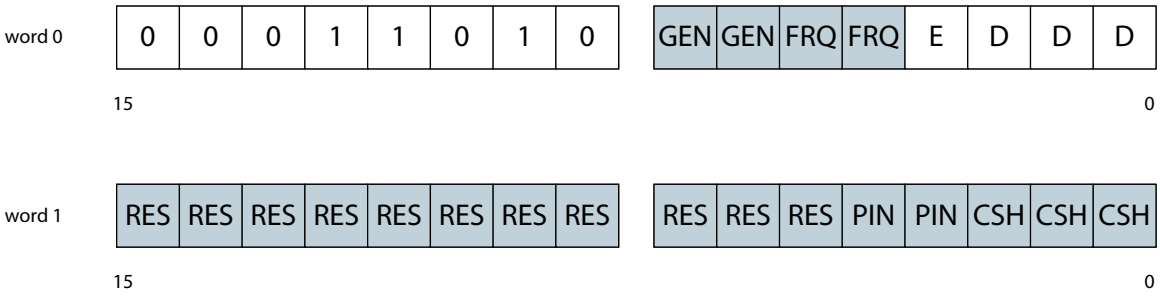

Figure B.28: Prepare Class Instruction

Description

This opcode creates a class from an array of words. The array of words is held in a destination register. Certain languages may require class verification when the class is created.

Required Attributes

All of the bits in the attribute byte are used for required attributes. The DDD bits specify the destination reference register that will hold the pointer to the new object. The E bit specifies whether the instruction has an additional word's worth of informational attributes.

Informational Attributes

Twenty bits can be used to hold informational attributes.

FRQ The two FRQ bits specify how frequently the class will be accessed and may provide information that is helpful to the cache. If FRQ is 00, then no access information is available. If FRQ is 01, then the class will be accessed infrequently. If FRQ is 10, then the class will be accessed frequently. If FRQ is 11, then the class will be accessed extremely frequently.

GEN The two GEN bits specify how long the class will be needed, and provides a hint to the virtual machine about possibly unloading the class in the future. If GEN is 00, then no generational information is provided. If GEN is 01, then the class will only be around for a short period of time. If GEN is 10, then the class will be around for a long period of time. If GEN is 11, then the class will never be unloaded from the virtual machine.

INR The INR bit specifies whether this class is an inner class. If IVR is clear, the method is not an inner class. If IVR is set, the method overwrites is an inner class of another class and objects of this class inherit the appropriate names, methods, and variables.

PIN The two PIN bits contain information about object pinning; the system can use this information to prevent the garbage collector from moving a particular class so pointer operations can be performed. If PIN is 00, then no pinning information is available. If PIN is 01, then the class should be pinned in place. If PIN is 10, then the class should be pinned at the location specified by the next four extension words. If PIN is 11, then the class should be unpinned and can be moved by the garbage collector.

CSH The three CSH bits indicate another reference register with which this class will be frequently used. The virtual machine should place the class in such a way so that it does not create cache conflicts with the class in this register.

RES The remaining eleven bits are reserved for future extensions to the virtual machine.

Extension Words

One, but future implementations may use the extension bit to add additional information attributes.

Exceptions

Execution of this instruction can result in the following exceptions:

InvalidClassException If the array of words is not in the class file format.

B.29 BLOCK START [SBK]

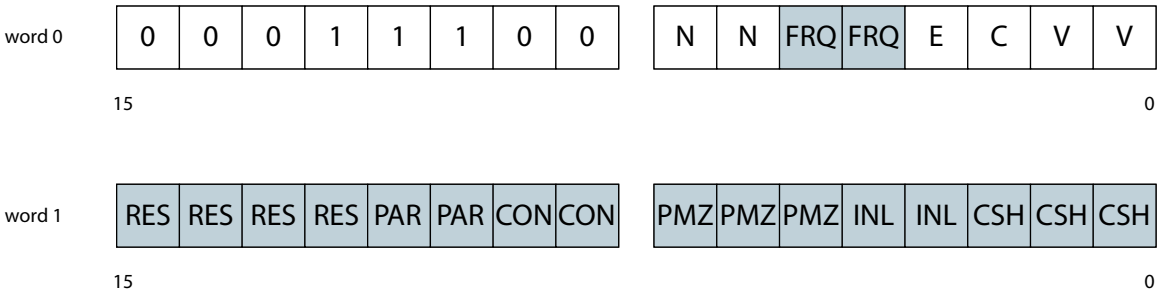

Figure B.29: Block Start Instruction

Description

This opcode signals the beginning of a block: a group of opcodes, a native method, or a library call which can be bound to a ++VM opcode. Blocks must be terminated using the block end opcode.

Required Attributes

This opcode has three required attributes. The NN bits specify the source of information for this block. The E bit specifies whether the instruction has an additional word's worth of informational attributes. The C bit specifies whether this block is critical to code execution. The V bits specify whether this block can be superceded by a block that is defined later.

The NN bits specify the composition of instructions in the block. If the NN bits are 00, then the block consists entirely of ++VM opcodes. If the NN bits are 01, an extension word holds a ten-bit index into the constant pool for the name of the library, and a six-bit index in that library for the function; this library function will consist of ++VM opcodes. If the NN bits are 10, then the block consists entirely of native instructions, and an extension word will specify the number of opcode words to parse in order to reach the end block opcode. If the NN bits are 11, an extension word holds a ten-bit index into the constant pool for the name of the library, and a six-bit index in that library for the function; this library function will consist of native instructions.

If the C bit is clear, this block is critical and the method should throw an exception if this block cannot be understood. If C is not set, the block is optional and can be ignored if not understood. If VV bit is 00, the opcode to which this block is bound cannot be re-bound to another block, and the virtual machine should throw an exception. If VV is 01, the opcode can be re-bound later in this method. If VV is 10, this opcode can be re-bound later in this class. If VV is 11, this opcode can be re-bound anywhere, including in other classes.

Informational Attributes

When starting a block, there are eighteen bits available for informational attributes.

FRQ The two FRQ bits specify how frequently the code block will be accessed and may provide information that is helpful to the cache. If FRQ is 00, then no access information is available. If FRQ is 01, then the code block will be accessed infrequently. If FRQ is 10, then the code block will be accessed frequently. If FRQ is 11, then the code block will be accessed extremely frequently.

PAR The two PAR bits specify whether the block can be parallelized over multiple threads. If PAR is 00, no information about block parallelization is provided. If PAR is 01, then the block cannot be parallelized using multiple threads due to possible side effects. If PAR is 10, the block can be parallelized over multiple threads. If PAR is 11, the virtual machine is strongly encouraged to parallelize the block.

CON The two CON bits specify whether the inputs to the block are constant. If CON is 00, then no information about block input is available. If CON is 01, then the block input will often fluctuate. If CON is 10, then block inputs will often be constants. If CON is 11, then block inputs will always be constant.

PMZ The three PMZ bits specify an optimization level. If PMZ is 000, then no optimization information is known. If PMZ is 001, this block should be interpreted and binary translation should not be used. The remaining values of PMZ specify an increasingly sophisticated optimization level that should be applied to the block, with a PMZ value of 111 specifying the greatest level of optimization.

INL The two INL bits specify whether this block should be inlined wherever it is called. If INL is 00, no inlining information is provided. If INL is 01, the block should not be inlined and should always be looked up in a method table. If

INL is 10, the block should be inlined into a method body where it is frequently used. If INL is 11, the block should be inlined into a method wherever it is used.

CSH The three CSH bits indicate another reference register with which this block will be frequently called. The virtual machine should place the block in such a way so that it does not create cache conflicts with the reference in this register.

RES The remaining four bits are reserved for future extensions.

Extension Words

One extension word holds the majority of the informational attributes available on this opcode. An additional word may be necessary to hold a constant pool offset or a length value, depending upon the value of the second N bit. Future implementations may use the extension bit to add additional information attributes.

Exceptions

Execution of this instruction can result in the following exceptions:

BlockDefinitionException If the block cannot be defined

BlockRedefinitionException If the block cannot be redefined

B.30 BLOCK END [EBK]

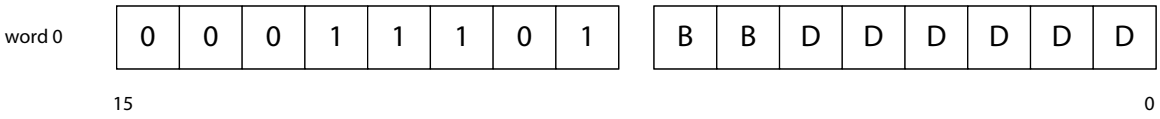

Figure B.30: Block End Instruction

Description

This opcode ends a block. The block can then be bound to an opcode at the method level, the block level, or the thread level, though it is possible not to assign a block to an opcode and simply use a block as a way of applying block attributes to a collection of opcodes.

Required Attributes

This opcode has two required attributes. The BB bits specify the scope to which the opcode should be bound. The SSSSSS bits specify the specific opcode to which the block should be bound.

If BB is 00, the block should not be bound to an opcode and the SSSSSS bits should be ignored. If BB is 01, the block should be bound to opcode 01SSSSSS (64-127) at the method level. If BB is 10, the block should be bound to opcode 10SSSSSS (128-191) at the class level. If BB is 11, the block should be bound to opcode 11SSSSSS (192-255) at the thread level.

Informational Attributes

No informational attributes are associated with this instruction.

Exceptions

Execution of this instruction can result in the following exceptions:

BlockDefinitionException If the block cannot be defined

BlockRedefinitionException If the block cannot be redefined

Programming Notes

This opcode must be matched with the most recent start block opcode.

B.31 THROW AN EXCEPTION [TRW]

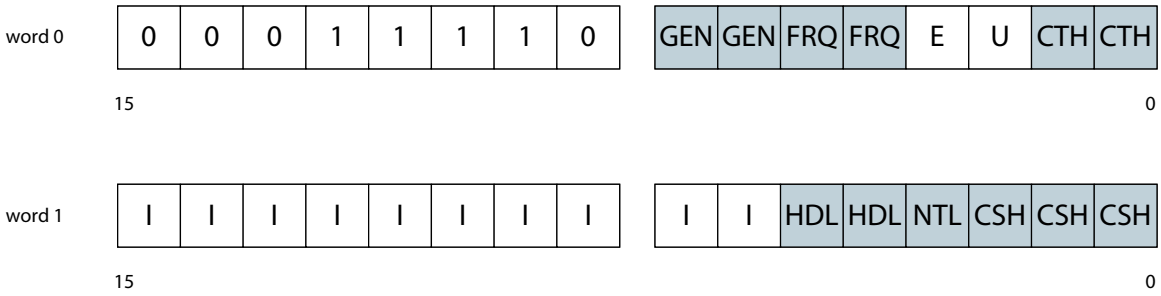

Figure B.31: Throw Exception Instruction

Description

This opcode creates an exception. The class of the exception to create is specified by a constant pool index. The exception is held in a machine register specialized for exceptions, the this exception register. The exception must be caught by an exception handler somewhere in the call stack; otherwise, the thread that threw the exception will terminate with an error.

Required Attributes

This opcode has three required attributes. The U bit specifies whether the this exception is uncatchable; if set, this exception cannot be caught by an exception handler. The E bit specifies whether the instruction has an additional word's worth of informational attributes. Ten bits of the second instruction word specify an index in the constant pool, the index of the exception to create.

Informational Attributes

Six bits in the first instruction word, and six bits in the second instruction word, are available for informational attributes.

GEN The two GEN bits specify how long the exception will live, and provides a hint to the virtual machine about garbage collection. If GEN is 00, then no generational information is provided. If GEN is 01, then the exception will be

around for a short period of time. If GEN is 10, then the exception will be around for a long period of time. If GEN is 11, then the exception will be nearly permanent. The 10 and 11 settings should be used rarely.

FRQ The two FRQ bits specify how frequently the exception will occur. If FRQ is 00, then no prediction information is available. If FRQ is 01, then this exception is unlikely to happen. If FRQ is 10, then this exception will happen frequently. If FRQ is 11, this exception will happen very frequently.

CTH The two CTH bits specify where the exception will likely be caught. If CTH is 00, than no information is available about the location of the handler. If CTH is 01, then the exception will be caught by an extension handler for this method. If CTH is 10, then the exception will be caught in the method that called this method. If CTH is 11, then this method will not be caught and will cause the thread to terminate.

HDL The two HDL bits specify how the exception will be caught. If HDL is 00, then no catching information is provided. If HDL is 01, then the handler that catches this exception will be looking for exceptions of the specified class. If HDL is 10, then the handler that catches this exception will be looking for the direct superclass of this exception type. If HDL is 11, then the handler will be looking for any superclass of this exception type.

NTL The NTL bit indicates whether the exception should be initialized. If NTL is clear, then the exception should be initialized. If NTL is set, then it is possible not to initialize the exception since only its class matters.

CSH The three CSH bits indicate another reference register with which this exception will be frequently used. The virtual machine should place the exception object in such a way so that it does not create cache conflicts with the reference in this register.

Extension Words

One, though future extensions may set the extension bit and use an additional extension word for attributes.

B.32 CATCH AN EXCEPTION [CAT]

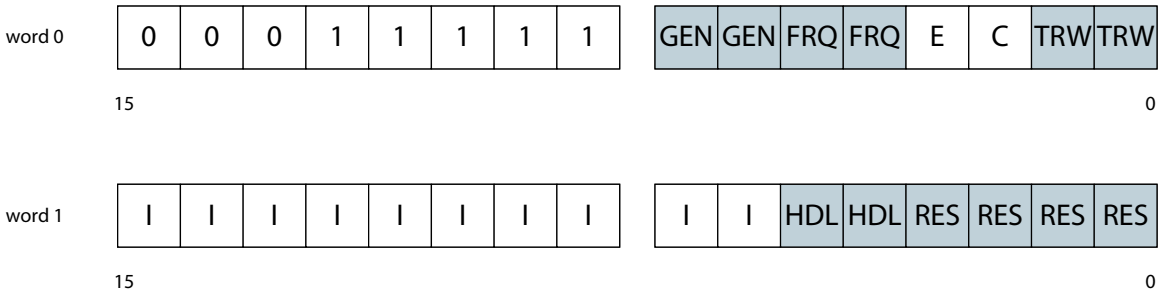

Figure B.32: Catch Exception Instruction

Description

This opcode creates an exception handler. The class of the exception to catch is specified by a class in the instruction pool. The behavior of this opcode depends upon the type of exception handling scheme used. If a stack unwinding scheme is used, this opcode will prepare an exception table for the method. If a stack cutting scheme is used, this opcode will set up an exception handler in the exception handler register.

Required Attributes

This instruction has three required attributes. The C bit specifies whether a stack unwinding or a stack cutting scheme should be used. The E bit specifies whether the instruction has an extra extension word. Ten bits of the second instruction word specify an index in the constant pool, the index of the exception to catch.

If C is 0, then stack unwinding should be used as exceptions will be relatively rare and the exception handler register will not be used. If C is 1, then stack cutting should be used and the exception handler should be placed in the exception handler register.

Informational Attributes

Six bits in the attribute byte, and six bits in the extension word, can be used to hold informational attributes.

GEN The two GEN bits specify how long the exception handler will be active here, and provides a hint to the virtual machine about when the handler will be overridden by another handler of the same type. If GEN is 00, then no overriding information is provided. If GEN is 01, then the handler will be overridden in the near future. If GEN is 10, then the handler will be around for a long time. If GEN is 11, then the handler will be nearly permanent.

FRQ The two FRQ bits specify how frequently the exception will be caught here. If FRQ is 00, then no prediction information is available. If FRQ is 01, then exception catching will rarely happen here. If FRQ is 10, then exception catching will happen frequently. If FRQ is 11, then exception catching will happen very frequently.

TRW The two TRW bits specify where the caught exception will be thrown. If TRW is 00, than no throwing information is available. If TRW is 01, then the exception will be thrown in this class. If TRW is 10, then the exception will be thrown from a method directly called by this class. If TRW is 11, then the exception will be thrown by a method deep in the call stack.

HDL The two HDL bits specify the type of exception to look for. If HDL is 00, then no catching information is provided. If HDL is 01, then the handler should look for exceptions of the specified class. If HDL is 10, then the handler should anticipate catching subclasses of this exception type. If HDL is 11, then the handler will be looking for interface implementations of this exception type.

RES The remaining four bits are reserved for future extensions to the virtual machine.

Extension Words

One, though future extensions may set the extension bit and use an additional extension word for attributes.

B.33 COMPARE DATA VALUES [CPD]

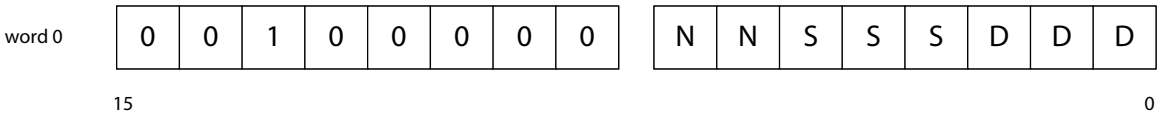

Figure B.33: Compare Data Instruction

Description

This opcode compares two data values stored in data registers, the stack, the instruction stream, or the constant pool. This instruction explicitly sets the bits in the condition control register. The type of comparison operation performed depends upon the tags of both values.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits indicate the destination register, though this register is unchanged during opcode execution. The five remaining bytes indicate the opcode source.

If NN is 00, the three SSS indicate a source register. If NN is 01, the three SSS bits are a small signed constant between -4 and +3. If NN is 10, the SSS bits are an index to one of the first eight entries in the class's constant pool. If NN is 11, then the SSS bits signify that the source is one of eight special cases.

In the case where NN is 11, if SSS is 000, then one extension word holds a ten-bit index to the source value in the constant pool. If SSS is 001, then the source value is found directly in one subsequent extension word. If S is 010, then the source value is found directly in two extension words, with the most significant word first. If SSS is 011, then the source value is found directly in four extension words. If SSS is 100, then the source is located at an offset from the this class pointer, an the offset is held in one extension word. If SSS is 101, then the source is located at an offset from the this class pointer, and the offset is held in one extension word. If SSS is 110, then the source is the top value of the stack. The SSS value 111, where NN is 11, is reserved for future use in the virtual machine and should not be used.

Extension Words

One, two or four extension words will be necessary if specified as a source location. The standard opcode does not need additional extension words.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

B.34 COMPARE REFERENCE VALUES [CPR]

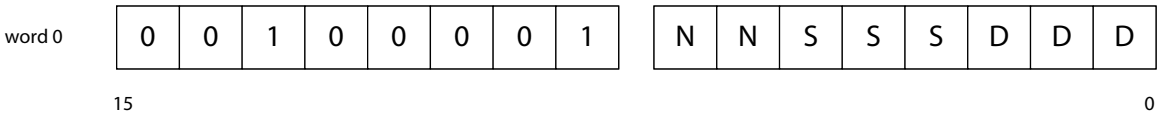

Figure B.34: Compare Reference Instruction

Description

This opcode compares two reference values from reference registers or the stack. This instruction performs a pure equality operation and is true if and only if the two objects are stored at the same location in memory. This instruction explicitly sets the bits in the condition control register.

Required Attributes

All of the bits in the attribute byte are required. The three least significant bits of the attribute byte, the bits labeled DDD, indicate the destination register. The five remaining bytes indicate the opcode source.

If NN is 00, the three SSS indicate a source register. If NN is 01, the source is the null object and the SSS bits are ignored. The NN value 01 is reserved and should not be used. If NN is 11, then the source comes from the top of the stack and the SSS bits are ignored.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags on the two values are not equal

B.35 COMPARE BOUNDS DATA [CBD]

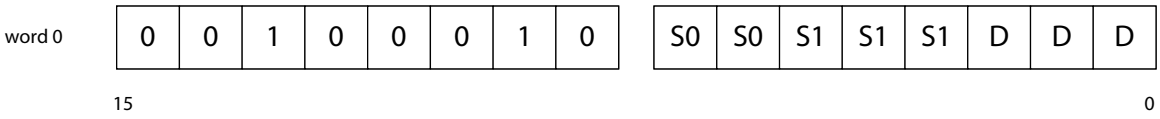

Figure B.35: Compare within Bounds for Data Instruction

Description

This opcode ensures that a certain value is greater than or equal to one argument and strictly less than another. All three values must be data types.

If the comparison succeeds and the value is within the bounds, the z bit is set in the condition code register. If the comparison fails because the number exceeds the upper bound, the n and z bits are cleared, and if it fails because the number is less than the lower bound, the n bit is set and the z is cleared. If the comparison fails for any other reason, both the n and z bits are set.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination data register. The source S1 specifies any one of the eight data registers, while the source S0 specifies one of the four low data registers (%d0 to %d3).

When this instruction is executed, DDD is the testing value, S0 is the lower bound, and S1 is the upper bound.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags of all three operands are not equal

B.36 COMPARE BOUNDS REFERENCE [CBR]

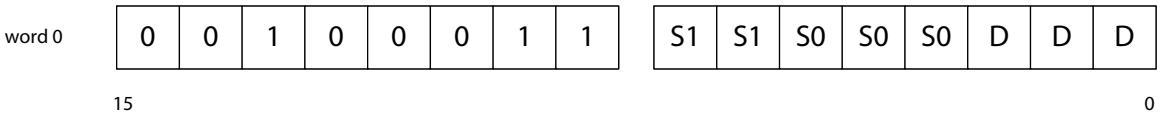

Figure B.36: Compare within Bounds for Reference Instruction

Description

This opcode ensures that a certain value is greater than or equal to one argument and strictly less than another. All three values must be reference types.

If the comparison succeeds and the value is within the bounds, the z bit is set in the condition code register. If the comparison fails because the number exceeds the upper bound, the n and z bits are cleared, and if it fails because the number is less than the lower bound, the n bit is set and the z is cleared. If the comparison fails for any other reason, both the n and z bits are set.

Required Attributes

All of the bits in the attribute byte are required. The DDD bits on the first instruction word indicate the destination reference register. The source S0 specifies any one of the eight reference registers, while the source S1 specifies one of the four low reference registers (%d0 to %d3).

When this instruction is executed, DDD is the testing value, S0 is the lower bound, and S1 is the upper bound.

Exceptions

Execution of this instruction can result in the following exceptions.

TagException If the tags of all three operands are not equal

Programming Notes

This instruction should be used with care, as the garbage collector may move reference values.

B.37 CONDITIONAL BRANCH [BNZ]

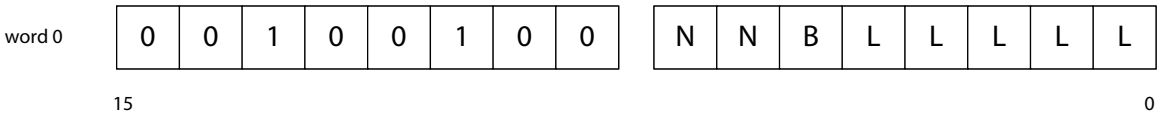

Figure B.37: Conditional Branch Instruction

Description

This opcode branches on various values of the n, z, and v bits of the condition code register. If the correct n and z bits are set, the program counter is changed to the specified value; otherwise, execution continues with the next instruction. The branch offset is held in the instruction word or in an additional extension word.

ALW The ALW instruction, NNB = 000, performs an unconditional branch.

NOP The NOP instruction, NNB = 001, serves as a no-op instruction. No branch is performed and the jump amount is ignored.

BEQ The BEQ instruction, NNB = 010, performs a branch equal if the z bit is set.

BNE The BNE instruction, NNB = 011, performs a branch not equal if the z bit is not set.

BLT The BLT instruction, NNB = 100, performs a branch less than if the n bit is set.

BGE The BGE instruction, NNB = 101, performs a branch greater than or equal if the n bit is not set.

BLE The BLE instruction, NNB = 110, performs a branch less than or equal if either the n or the z bits are set.

BGT The BGT instruction, NNB = 111, performs a branch greater than if neither the n nor z bits are set.

Required Attributes

All of the bits in the attribute byte are required. The NN bits select the conditions on which to jump. The B bit determines whether to branch if the condition code is set or to branch if the condition code is not set. The LLLLL bits are a signed integer that indicates the branch amount from the the first word of this opcode.

If NNB is 000, an ALW is performed. If NNB is 001, a NOP is performed. If NNB is 010, a BEQ is performed. If NNB is 011, a BNE is performed. If NNB is 100, a BLT is performed. If NNB is 101, a BGE is performed. If NNB is 110, a BLE is performed. If NNB is BGT, a BGT is performed.

If LLLLL is not 00000, the virtual machine will treat that value as a five-bit signed integer and add it to the current program counter. Since a branch instruction back on itself is superfluous, If LLLLL is 00000 then one extension word holds a sixteen-bit signed offset to the branch location.

Informational Attributes

None, though predictive branching is implied. The virtual machine will assume that the test will succeed and will load the instruction pipeline appropriately, even though no informational attribute is expressly used.

Extension Words

If the jump bits are 00000, then one extension word is used to hold a 16-bit jump index.

B.38 CONDITIONAL BRANCH [BXC]

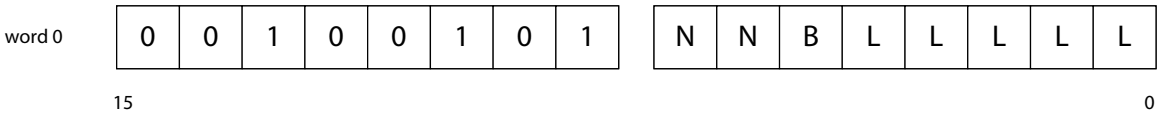

Figure B.38: Conditional Branch Instruction

Description

This opcode branches on various exception bits in the condition code register. If the correct bits are set, the program counter is change to the specified value; otherwise, execution continues with the next instruction. The branch offset is held in the instruction word or in an additional extension word.

Required Attributes

All of the bits in the attribute byte are required. The NN bits select the conditions on which to jump. The B bit determines whether to branch if the condition code is set or to branch if the condition code is not set. The LLLLL bits are a signed integer that indicates the branch amount from the the first word of this opcode.

If NNB is 000, the virtual machine will branch if the overflow bit is set. If NNB is 001, a branch will occur if the overflow bit is not set. If NNB is 010, a branch will occur if the underflow bit is set. If NNB is 011, a branch will occur if the underflow bit is not set. If NNB is 100, a branch will occur if the exception bit is set. If NNB is 101, a branch will occur if the exception bit is not set. If NNB is 110, a branch will occur if the future bit is set. If NNB is 111, a branch will occur if the future bit is not set.

If LLLLL is 00000, then one extension word holds a sixteen-bit signed offset to the branch location. If LLLLL is not 00000, the virtual machine will treat that value as a five-bit signed integer and add it to the current program counter.

Informational Attributes

None, though predictive branching is implied. The virtual machine will assume that the test will succeed and will load the instruction pipeline appropriately, even though no informational attribute is expressly used.

Extension Words

If the jump bits are 00000, then one extension word is used to hold a 16-bit jump index.

B.39 CALL METHOD [CAL]

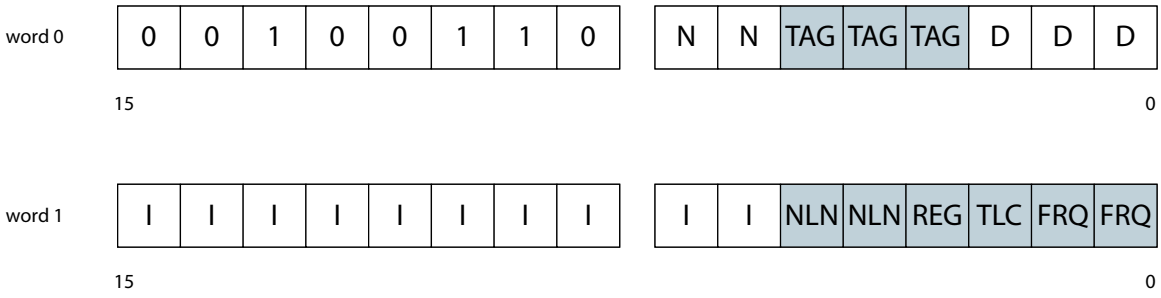

Figure B.39: Invoke Method Instruction

Description

This opcode calls a method. Methods can be called in four forms: statically, dynamically, wound, or via a closure. A new frame is created to hold the method, if applicable. The index of the method to call is passed via a method index. The method's return values will be placed in registers %d6 and %d7 if they are data values or in registers %r6 and %r7 if they are reference values.

Required Attributes

The call instruction has four required attributes. The E bit specifies whether the instruction has an additional word's worth of informational attributes. The NN bits specify the type of method invocation to perform. The RRR bits specifies whether the method returns data values, reference values, or both. Ten bits of the second instruction word specify an index into the constant pool, the index of the method to call.

If NN is 00, this invocation is a static method invocation and the %to should be ignored. If NN is 01, this invocation is virtual and the %to should be used. If NN is 10, this invocation is a wound call and should be called first on the ancestors of this class before calling it on this class. If NN is 11, the method is called via a closure, the ten-bit index is ignored, and the TAG attribute bits are used to designate a data register that holds the closure index.

If RRR is 000, the method returns void. The RRR value of 111 is reserved. If RRR is 010, the method returns one data value in %d6. If RRR is 011, the method

returns two data values in %d6 and %d7. If RRR is 100, the method returns one reference value in %r6. If RRR is 101, the method returns two data values in %r6 and %r7. If RRR is 110, the method returns one data and one reference value, in %d6 and %r6 respectively. If RRR is 111, the method returns four values in %d6, %d7, %r6, and %r7.

Informational Attributes

Three bits in the first instruction word, and six bits in the second, hold informational attributes.

FRQ The two FRQ bits contain information about how frequently this method will be called in the future; the system may be able to use this information to predict program behavior more efficiently. If FRQ is 00, then no calling information information is known. If FRQ is 01, then the method will not be called frequently. If FRQ is 10, then the method will be called relatively frequently in the near future. If FRQ is 11, then the method will be called extremely frequently.

NLN The two NLN bits contain information about whether the method should be inlined. If NLN is 00, then no inlining information is available. If NLN is 01, then this method should be inlined wherever it is called. If NLN is 10, then the method should not be inlined where it is called. If NLN is 11, then the method will be expensive to optimize.

TLC The TLC bit contains whether the method should be invoked using a tail call. If TLC is clear, the method should be called normally. If TLC is set, then the method should be called in the same frame as the invoking frame.

TAG The three tag bits specify the tag of the reference on which this method will be called. This attribute can be used to optimize for the creation of normal objects, objects with code, futures, and pointers. This attribute is only available to non-closure methods.

Extension Words

One extension word holds the index and information attribute values.

B.40 RETURN FROM METHOD CALL [RTN]

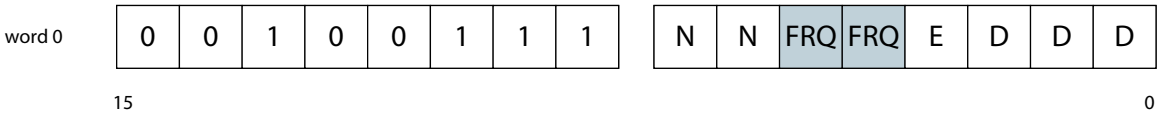

Figure B.40: Return Instruction

Description

This opcode returns from a method call, returning a destination register, a reference register, or void. The method that executed the bytecode is terminated, its frame is destroyed, and execution proceeds in the outer method. The method's return values must be placed in registers %d0 and %d1 if they are data values or in registers %r0 and %r1 if they are reference values.

Required Attributes

This instruction has three required attributes. The E bit specifies whether the instruction has an extra extension word. The NN bits specify the type of return to perform. The R bit specifies the number of values to return.

If NN is 00, the method returns void and R should be set to zero. If NN is 01 and R is 0, the method returns the value in %d0. If NN is 01 and R is 1, then the method returns both the values in %d0 and %d1. If NN is 10 and R is 0, the method returns the value in %r0. If NN is 10 and R is 1, then the method returns the values in %r0 and %r1. If NN is 11 and R is 0, the method returns values in both %d0 and %r0. If NN is 11 and R is 1, the method returns the values in %d0, %d1, %r0, and %r1.

Informational Attributes

Two bits are available for informational attributes.

FRQ The two FRQ bits specify how frequently this return will be executed. If FRQ is 00, then no prediction information is available. If FRQ is 01, then this return will rarely be executed. If FRQ is 10, then this return will be frequently executed. If FRQ is 11, then this return will always be executed. This information may help the system establish traces through the method.

TRN The two TRN bits specify whether method execution to this point has required using values in registers %d2 to %d7 or registers %r2 to %r7, possibly allowing future calls of this method to avoid a register window rotation. If TRN is 00, then neither the data registers nor the reference registers have been manipulated and no window turn is needed. If TRN is 01, then only data registers have been manipulated and the reference register window need not be turned. If TRN is 10, then only reference registers have been manipulated and the data register window need not be turned. If TRN is 11, both data and reference values have been manipulated and so it will always be necessary to shift the register window.

Extension Words

None, but future implementations may use the extension bit to add additional information attributes.

B.41 ENTER A MONITOR [MEN]

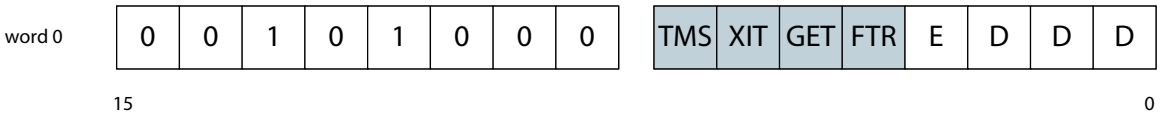

Figure B.41: Enter Monitor Instruction

Description

This opcode grabs a thread monitor. If the grab is successful, the thread locks the object and continues code execution; if the grab is unsuccessful, it will either wait for the lock to become available or it will create a future for the object.

Required Attributes

This instruction has two required attributes. The three least significant bits of the first instruction word, DDD, specify the object whose monitor should be grabbed. The E bit specifies whether the instruction has an extra extension word.

Informational Attributes

Four bits in the original instruction are available for informational attributes.

TMS The TMS bit specifies how many times to lock a given object. If TMS is clear, the object is locked once. If TMS is set, the object is locked as many times as possible.

XIT The XIT bit predicts how the monitor will be released. If XIT is clear, the monitor will probably be released through the execution of a monitor exit opcode. If XIT is set, the monitor will probably released by an exception handler.

GET The GET bit specifies whether or not the monitor grab will be successful and can be used for pipelining. If GET is clear, then the thread will likely be able to grab the monitor and will not have to be suspended. If GET is set, then the thread will be unlikely to be able to grab the monitor.

FTR The FTR bit specifies whether a future should be created instead of suspending the thread. If FTR is set, the thread should suspend and no future will be created. If FTR is set, then execution should continue but the monitored object should be treated as a future.

Extension Words

None, though future extensions may set the extension bit and use an additional extension word for attributes.

B.42 EXIT A MONITOR [MEX]

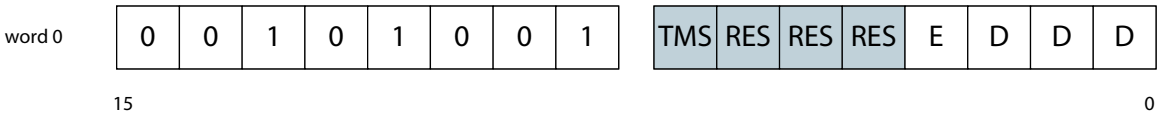

Figure B.42: Monitor Exit Instruction

Description

This opcode leaves a thread monitor. If the release is successful, the thread decreases the lock on the object and continues code execution; if the object is not locked by this thread, an exception is thrown.

Required Attributes

This instruction has two required attributes. The three least significant bits of the first instruction word, DDD, specify the object whose monitor should be grabbed. The E bit specifies whether the instruction has an extra extension word.

Informational Attributes

Four bits in the original instruction are available for informational attributes.

TMS The TMS bit specifies how many times to unlock a given object. If TMS is clear, the object is unlocked once. If TMS is set, all locks on the object are released.

RES The remaining three bits are reserved for future extensions.

Extension Words

None, though future extensions may set the extension bit and use an additional extension word for attributes.

B.43 YIELD PROCESSOR CONTROL [YLD]

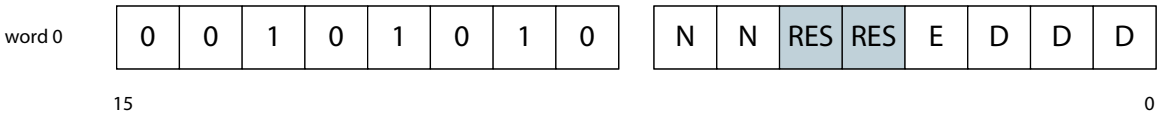

Figure B.43: Yield Processor Instruction

Description

This opcode tells a thread to suspend itself so another thread can run. This may be useful to multithreaded programs.

Required Attributes

This instruction has two required attributes. The DDD bits specify the thread to which control should be passed. The E bit specifies whether the instruction has an extra extension word. The NN bits specify the type of yield to perform.

If NN is 00, control should be yielded to the system; the DDD register should be ignored. If NN is 01, control should be yielded to the system for one quanta and then back to this thread; the DDD register should be ignored. If NN is 10, control should be yielded to the DDD thread without constraints. If NN is 11, control should be yielded to the DDD thread for one quanta and then back to this thread.

Informational Attributes

Two bits in the original instruction are available for informational attributes.

RES Two bits are reserved for future extensions.

Extension Words

None, though future extensions may set the extension bit and use an additional extension word for attributes.

B.44 SYNCHRONIZATION POINT [SYN]

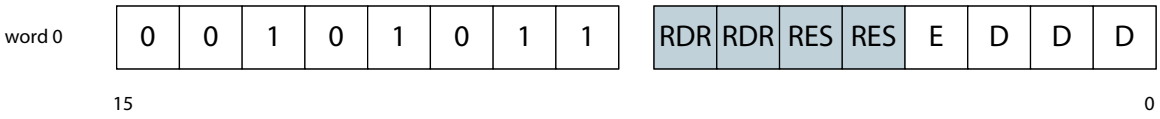

Figure B.44: Synchronization Point Instruction

Description

This opcode creates a synchronization point within the code. This is done by synchronizing on a specific checkpoint object; when all threads have synchronized on the object, they are all released. This provides a way to synchronize across multiple threads.

Required Attributes

This instruction has two required attributes. The DDD bits specify the synchronization object. The E bit specifies whether the instruction has an extra extension word.

Informational Attributes

Four bits are available for informational attributes.

RDR The RDR bits specify the probable order in which a thread will synchronize. If RDR is 00, no synchronization information is available. If RDR is 01, this thread will probably be the first to synchronize. If RDR is 10, this thread will probably be in the middle of the pack. If RDR is 11, this thread will probably be the last to synchronize.

RES The remaining two bits are reserved for future extensions.

Extension Words

None, though future extensions may set the extension bit and use an additional extension word for attributes.

B.45 ANNOTATION [ANO]

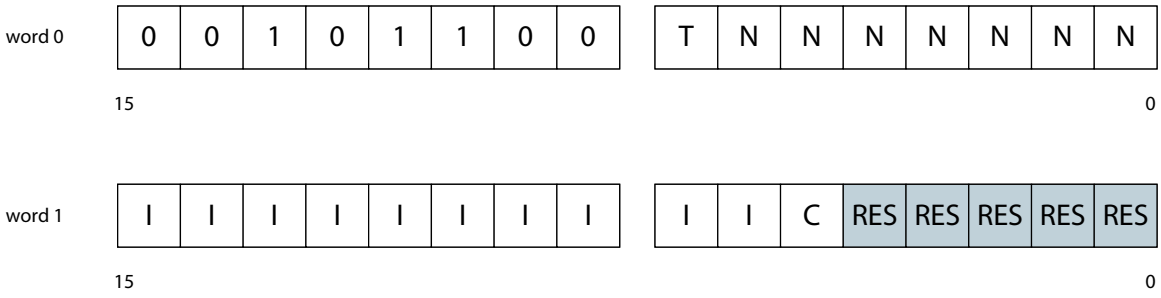

Figure B.45: Annotation Instruction

Description

This opcode defines a code annotation, a way of providing additional information about a set of opcodes. An annotation consists of an annotation header opcode an a specified number of annotation words. The body of the annotation contains meta-information about the code and can be used by the virtual machine to improve code performance.

Annotations are defined by the implementer or other language designers, and the precise usage of an annotation is left to the compiler writer and implementation designer. If an annotation is not understood by the virtual machine, it is ignored.

Required Attributes

All the bits in the attribute bytes are required attributes. The NNNNNNN bits specify an integer. The T bits specifies whether the integer is a constant or an index to a constant.

If T is clear, the annotation length is described by the seven bit integer NNNNNNN. If T is set, the annotation length is held in the constant pool at the seven-bit index NNNNNNN.

The next word contains information about the name and type of the annotation. This is held by a ten-bit index into the instruction pool. The length of this word is included in the attribute length. The C bit specifies whether the annotation is critical.

If the C bit is clear, this block is critical and the method should throw an exception if this block cannot be understood. If C is not set, the block is optional and can be ignored if not understood.

Informational Attributes

Five bits are available for informational attributes.

RES The remaining five bits are reserved for future extensions.

Extension Words

The specified number of extension words will be attached to the annotation opcode. These extension words represent the annotation propper, and will be skipped if the opcode is not understood.

B.46 LOOP [LUP]

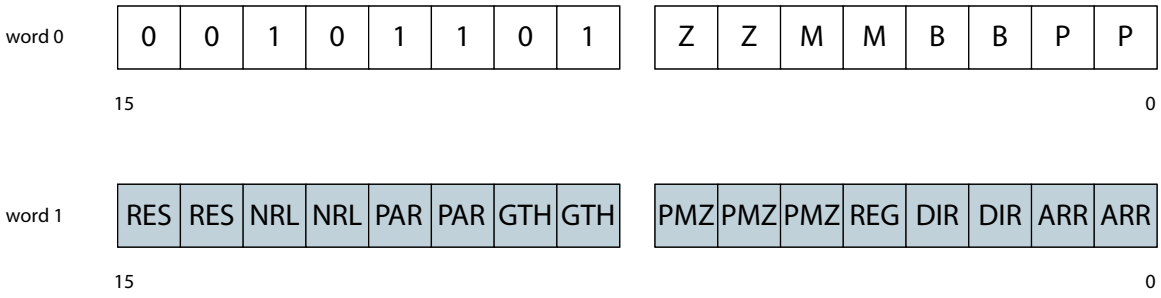

Figure B.46: Loop Instruction

Description

This opcode is a loop annotation, an annotation common enough to be incorporated into the standard opcode set. A loop is divided into four components: the initialization component, which is executed before the loop body; the comparison component, which is the test that is performed at the top of the loop; the body component, which is the repeated code in the loop; and the continue component, which changes the loop variable. The loop opcode specifies which chunks of code make up these components, as well as specifying additional information which may help improve the efficiency of loop execution.

Required Attributes

There are eight required attributes in the loop instruction. The ZZ bits specify the length of the initialization component, the MM bits specify the length of the comparison component, the BB bits specify the length of the body component, and the PP bits specify the length of the continue component.

All of these required attribute parts have the same form. If the bits have the pattern 00, one extension word holds the component length as a sixteen-bit integer. If the bits are 01, one extension word holds a ten-bit index into the constant pool that indicates the length of the component. If the bits are 01, no extension word is needed since a block opcode is used to hold the specified component. The value 11 is reserved for future extensions.

Informational Attributes

Sixteen bits in an extension word are used to hold informational attributes.

NRL The two NRL bits specify whether this loop should be unrolled. If NRL is 00, no unrolling information is provided. If NRL is 01, the block should not be unrolled. If NRL is 10, the loop should be unrolled if it is frequently executed. If NRL is 11, the loop should be unrolled regardless of the number of times it is executed.

PAR The two PAR bits specify whether the loop can be parallelized over multiple threads. If PAR is 00, no information about loop parallelization is provided. If PAR is 01, then the loop cannot be parallelized using multiple threads due to possible side effects. If PAR is 10, the loop can be parallelized over multiple threads. If PAR is 11, the virtual machine is strongly encouraged to parallelize the loop.

GTH The two GTH bits predict how long this loop will last. If GTH is 00, then no loop length information is available. If GTH is 01, then the loop length will be short. If GTH is 10, then the loop length will be long. If A5 is 11, then the class will be infinite.

PMZ The three PMZ bits specify an optimization level. If PMZ is 000, then no optimization information is known. If PMZ is 001, this block should be interpreted and binary translation should not be used. The remaining values of PMZ specify an increasingly sophisticated optimization level that should be applied to the block, with a PMZ value of 111 specifying the greatest level of optimization.

REG The REG bit specifies whether the loop counter variable should be kept in a data register. If REG is clear, then no information is available. If REG is set, then the loop counter should be kept in a host register as much as possible.

DIR The two DIR bits specify the direction of the loop for caching purposes. If DIR is 00, no information about the direction of the loop counter is available for caching. If DIR is 01, then the loop counter moves incrementally upwards. If DIR is 10, the loop counter moves incrementally downwards. If DIR is 11, the loop counter moves around randomly and predicting counter behavior may be difficult.

ARY The two ARY bits specify whether an array is being iterated in the loop. If ARY is 00, then no information is available. If ARY is 01, then the counter is

not iterating over an array and is not an index into an array. If ARY is 10, then the counter is iterating over a single array. If ARY is 11, then the counter is iterating over multiple arrays.

RES The remaining two bits are reserved for future extensions.

Extension Words

Up to four extension words are needed to hold the lengths of the four components (initialization, comparison, body, and continuation) of the loop. Fewer words will be needed if the components are specified as blocks.

Bibliography

[ABC+] Ole Agsen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Holzle, John Maloney, Randall B. Smith, and David Ungar.

[Ash05] Elaine Ashton. Perl.org, 2005.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[BCF+99] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM Press.

[BDF+03] Paul Barham, Doris Dragovic, Keir Fraser, Stven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[BFHW75] J. D. Bagley, E. R. Floto, S. C. Hsieh, and V. Watson. Sharing data and services in a virtual machine system. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 82–88, New York, NY, USA, 1975. ACM Press.

[BS96] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computing Systems*, 14(1):80–107, 1996.

[CHL00] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–75, 2000.

[CK94] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, New York, NY, USA, 1994. ACM Press.

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[Cor05a] Apple Corporation. The brains behind apple's rosetta: Transitive. CNet News.com", 2005.

[Cor05b] Microsoft Corporation. Common language runtime overview, 2005.

[Cor05c] Transmeta Corporation. Tansmeta code morphing software. Crusoe Architecture, 2005.

[Cre81] Robert J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[CRP+05] Daniel Chaver, Miguel A. Rojas, Luis Pinuel, Manuel Prieto, Francisco Tirado, and Michael C. Huang. Energy-aware fetch mechanism: trace cache and btb customization. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 42–47, New York, NY, USA, 2005. ACM Press.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70, New York, NY, 1989. ACM Press.

[Dic73] Lloyd I. Dickman. Small virtual machines: A survey. In *Proceedings of the workshop on virtual computer systems*, pages 191–202, New York, NY, USA, 1973. ACM Press.

[Dik00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[Fat81] Richard J. Fateman. Views on transportability of lisp and lisp-based systems. In *SYMSAC '81: Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 137–141, New York, NY, USA, 1981. ACM Press.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Operating Systems Design and Implementation*, pages 137–151, 1996.

[FHN+04] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Reconstructing i/o. Technical Report UCAM-CL-TR-596, University of Cambridge, Computer Laboratory, August 2004.

[FM90] Marc Feeley and James S. Miller. A parallel virtual machine for efficient scheme compilation. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 119–130, New York, NY, USA, 1990. ACM Press.

[GH97] Bruce Greer and Greg Henry. High performance software on intel pentium pro processors or micro-ops to teraflops. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, New York, NY, USA, 1997. ACM Press.

[GM96] James Gosling and Henry McGilton. The java language environment. White Paper, 1996.

[Gol73] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM Press.

[GR80] L. J. Groves and W. J. Rogers. The design of a virtual machine for ada. In *SIGPLAN '80: Proceeding of the ACM-SIGPLAN symposium on Ada programming language*, pages 223–234, New York, NY, USA, 1980. ACM Press.

[GR83a] Adele Goldberg and David Robson. *Formal Specification of the Object in Memory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GR83b] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GTHR99] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 154–169, New York, NY, USA, 1999. ACM Press.

[Gum83] Peter H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983.

[HS04] Shiliang Hu and James E. Smith. Using dynamic binary translation to fuse dependent instructions. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 213, Washington, DC, USA, 2004. IEEE Computer Society.

[Hug97] Jim Hugunin. Python and java: The best of both worlds. 0, 1997.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM Press.

[Joh78] JohnMcCarthy. History of lisp. In *HOPL-1: The first ACM SIGPLAN conference on History of Programming Languages*, pages 217–223, New York, NY, USA, 1978. ACM Press.

[KDC03] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 71–84, June 2003.

[Kla00] Alexander Klaiber. The technology behind crusoe processors. White Paper of Transmeta Corporation, January 2000.

[Kum04] K. V. Seshu Kumar. When and what to compile/optimize in a virtual machine? *SIGPLAN Not.*, 39(3):38–45, 2004.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Mac05] Jikes RVM Virtual Machine. Class vm_javaheader. Jikes RVM Header Files and Documentation, 2005.

[Mal73] Efrem G. Mallach. On the relationship between virtual machines and emulators. In *Proceedings of the workshop on virtual computer systems*, pages 117–126, New York, NY, USA, 1973. ACM Press.

[Mar03] Alex Martelli. *Python in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[Mir87] Eliot Miranda. Brouhaha- a portable smalltalk interpreter. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 354–365, New York, NY, USA, 1987. ACM Press.

[Mit03] John Mitchel. *Concepts in Object-Oriented Languages*. Cambridge University Press, 2003.

[Nel79] Philip A. Nelson. A comparison of pascal intermediate languages. In *SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, pages 208–213, New York, NY, USA, 1979. ACM Press.

[NR302a] ECMA NR39/TG3. *Common Language Infrastructure*, 2002.

[NR302b] ECMA NR39/TG3. *Common Language Infrastructure*, 2002.

[OKN01] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. A study of exception handling and its dynamic optimization in java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 83–95, New York, NY, USA, 2001. ACM Press.

[Org03] Python Organization. Python library reference. Python Website, 2003.

[Org05a] Jython Organization. Jython documentation. Jython Web Site, 2005.

[Org05b] Parrot Organization. Parrotcode: Parrot documentation. Documentation Snapshot from the Parrot Source, 2005.

[Org06] Python Organization. The python programming language website, 2006.

[PD82] Steven Pemberton and Martin Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Prentice Hall, 1982.

[Pel04] Michel Pelletier. Python bytecode verification. Python Standards Track, 2004.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[SAP05] SAP. Startup supplies translation technology to apple. SAP info, 2005.

[SCP+02] Constantine P. Sapntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monika S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.

[SG96] Guy Steele and Richard Gabriel. The evolution of lisp. *History of Programming Languages - II*, pages 233–330, 1996.

[SGG05] Abraham Silberschatz, Greg Gange, and Peter Galvin. *Operating System Concepts 7th ed.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[SM79] Love H. Seawright and Richard A. MacKinnon. Vm/370 - a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[SN05] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kauffman Publishers, San Francisco, CA, USA, 2005.

[Sug02] Dan Sugalski. Parrot in detail. In *Yet Another Perl Conference*, 2002.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMWare workstation's hosted virtual machine monitor, 2001.

[Tai98] Antero Taivalsaari. Implementing a java tm virtual machine in the java programming language, 1998.

[Uea05] Rich Uhlig and Gil Neiger et al. Intel virtualization technology. *Computer Magazine*, 38(5):48–56", 2005.

[Wal02] Carl Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[WBC99] Allen Wirfs-Brock and Pat Caudill. Instantiations, Inc Paper, 1999.

[WCSG05] Andrew Whitaker, Richard Cox, Marianne Shaw, and Steven Gribble. Rethinking the design of virtual machine monitors. *Computer Magazine*, 38(5):57–62, 2005.

[Wir93] N. Wirth. Recollections about the development of pascal. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 333–342, New York, NY, USA, 1993. ACM Press.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[YvR01] Ka-Ping Yee and Guido van Rossum. Iterators. Python Standards Track, 2001.