1 The Java Virtual Machine

About the Spec

Format

This document describes the Java virtual machine and the instruction set. In this introduction, each component of the machine is briefly described. This introduction includes a description of the format used to present the opcode instructions. The next chapter is the instructions themselves.

Chapter 3 is the spec for the Java class file format, the binary file produced by the Java compiler. The file will contain information about the class, its fields, its methods, and the virtual machine code required to execute the methods.

Appendix A contains some instructions used internally on the LiveOak project for compiler optimization.

Purpose and Vision

The Java virtual machine specification has a purpose that is both like and unlike equivalent documents for other languages and abstract machines. It is intended to present an abstract, logical machine design free from the distraction of inconsequential details of any implementation. It does not anticipate an implementation technology, or an implementation host. At the same time it gives a reader sufficient information to enable implementation of the abstract design in a range of technologies.

However, the intent of the LiveOak project is to create a language and application that will allow the interchange over the Internet of "executable content," which will be embodied by compiled Java code. The project specifically does not want Java to be a proprietary language, and does not want to be the sole purveyor of Java language implementations. Rather, we hope to make documents like this one, and source code for our implementation, freely available for people to use as they choose.

This vision for LiveOak can only be achieved if the executable content can be reliably shared between different Java implementations. These intentions prohibit the definition of the Java virtual machine from being fully abstract. Rather, relevant logical elements of the design have to be made sufficiently concrete to enable the interchange of compiled Java code. This does not collapse the Java virtual machine specification to a description of an Java implementation; elements of the design that do not play a part in the interchange of executable content remain abstract. But it does force us to specify, in addition to the abstract machine design, a concrete interchange format for compiled Java code.

The Java Interchange Specification

The Java interchange specification must contain the following components:

- the instruction set syntax, including opcode and operand sizes and types, alignment and endianness
- the instruction set opcode values
- the values of any identifiers (e.g. type identifiers) in instructions or in supporting structures
- the layout of supporting structures that appear in compiled Java code (e.g. the constant pool)
- the Java object format (the .class file format).

In this version of the Java virtual machine specification, many of these have not yet been described, and are priorities for the next release of the document.

Abstractions Left to the Implementor

Elements of the design unrelated to the interchange of compiled Java code remain abstract, including:

- layout and management of the runtime data areas

- garbage collection algorithms, strategies and constraints
- the compiler, development environment, and runtime (apart from the need to generate and read valid compiled Java code)
- optimizations that can be performed once compiled Java code is received.

Components of the Virtual Machine

The Java virtual machine consists of:

- An instruction set
- A set of registers
- A stack
- A garbage-collected heap
- A method area

All of these are logical, abstract components of the virtual machine. They do not presuppose any particular implementation technology or organization, but their functionality must be supplied in some fashion in every Java system based on this virtual machine. The Java virtual machine may be implemented using any of the conventional techniques: e.g. bytecode interpretation, compilation to native code, or silicon.

The memory areas of the Java virtual machine do not presuppose any particular locations in memory or locations with respect to one another. The memory areas need not consist of contiguous memory. However, the instruction set, registers, and memory areas are required to represent values of certain minimum logical widths (e.g. the Java stack is 32 bits wide). These requirements are discussed in the following sections.

The Java Instruction Set

The Java instruction set is the assembly-language equivalent of an Java application. Java applications are compiled down to the Java instruction set just like C applications are compiled down to the instruction set of a microprocessor. An instruction of the Java instruction set consists of an *opcode* specifying the operation to be performed, and zero or more *operands* supplying parameters or data that will be used by the operation. Many instructions have no operands and consist only of an opcode.

The opcodes of the Java instruction set are always one byte long, while operands may be of various sizes.

When operands are more than one byte long they are stored in "big-endian" order — high order byte first. For example, a 16-bit parameter is stored as two bytes whose value is:

```
first_byte * 256 + second_byte
```

Operands that are larger than 8 bits are typically constructed from byte-sized quantities at runtime — the instruction stream is only byte-aligned and alignment of larger quantities is not guaranteed. (An exception to this rule are the *tableswitch* and *lookupswitch* instructions.) These decisions keep the virtual machine code for a compiled Java program compact and reflect a conscious bias in favor of compactness possibly at some cost in performance.

Primitive Data Types

The instruction set of the Java virtual machine interprets data in the virtual machine's runtime data areas as belonging to a small number of primitive types. Primitive numeric types include integer, long, single and double precision floating point, byte and short. All numeric data types are signed. Unsigned short exists for use as (Unicode) chars only. In addition, the *object* type is used to represent Java objects in computations. Finally, a small number of operations (e.g. the *dup* instructions) operate on runtime data areas as raw values of a given width without regard to type.

Primitive data types are managed by the compiler, not the compiled Java program or the Java runtime. In particular, primitive data are not necessarily tagged or otherwise discernible at runtime. The Java instruction set distinguishes operations on different primitive data types with different opcodes. For instance, *iadd*, *ladd*, *fadd* and *dadd* instructions all add two numbers, but operate on integers, longs, single floats and double floats, respectively.

Registers

The registers of the Java virtual machine maintain machine state during its operation. They are directly analogous to the registers of a microprocessor. The Java virtual machine's registers include:

- pc — the Java program counter
- optop — a pointer to the top of the Java operand stack
- frame — a pointer to the execution environment of the currently executing method
- vars — a pointer to the 0th local variable of the currently executing method

The Java virtual machine defines each of its registers to be 32 bits wide. Some Java implementations may not use all of these registers: e.g. a compiler from Java source to native code does not maintain pc.

The Java virtual machine is stack-based, so it does not define or use registers for passing or receiving parameters. This is again a conscious decision in favor of instruction set simplicity and compactness, and efficient implementation on host processors without many registers (e.g. Intel 486).

The Java Stack

The Java virtual machine is a stack-based machine, and the Java stack is used to supply parameters for operations, receive return values, pass parameters to methods, etc. An Java stack frame is Java's equivalent to the stack frame of a conventional programming language. It implements the state associated with a single method invocation. Frames for nested method calls are stacked on the method invocation stack.

Each Java stack frame consists of three components, although at any given time one or more of the components may be empty:

- the local variables
- the execution environment
- the operand stack

The size of the local variables and the execution environment are fixed on method call, while the operand stack varies as the method is being executed. Each of these components is discussed below.

Local Variables

Each Java stack frame has a set of local variables. They are addressed as indices from the vars register, so are effectively an array. Local variables are all 32 bits wide.

Long integers and double precision floats are considered to take up two local variables but are addressed by the index of the first local variable (e.g. a local variable with index n containing a double precision float actually occupies storage at indices n and n+1). 64-bit values in local variables are not guaranteed to be 64-bit aligned. Implementors are free to decide the appropriate way to divide long integers and double precision floats into the two registers.

Instructions are provided to load the value of local variables values onto the operand stack and store values from the operand stack into local variables.

Execution Environment

The execution environment is the component of the stack frame used to maintain the operations of the Java stack itself. It contains pointers to the previous frame as well as pointers to its own local variables and operand stack base and top. Additional per-invocation information (e.g. for debugging) belongs in the execution environment.

Exceptions

Each Java method has a list of *catch clauses* associated with it. Each catch clause describes the instruction range for which it is active, the type of exception that it is to handle and has a chunk of code to handle it. When an exception is tossed, the catch list for the current method is searched for a match. An exception matches a catch clause if the instruction that caused the exception is in the appropriate instruction range, and the thrown exception is a subtype of the type of exception that the catch clause handles.

If a matcing catch clause is found, the system branches to the handler. If no handler is found, the current stack frame is popped and the exception is raised again.

The order of the catch clauses in the list is important. The interpreter branches to the first matching catch clause.

Operand Stack

The operand stack is a 32 bit wide FIFO stack used to store arguments and return values of many of the virtual machine instructions. For example, the *iadd* instruction adds two integers together. It expects that the integers to be added are the top two words on the operand stack, pushed there by previous instructions. Both integers are popped from the stack, added, and their sum pushed back onto the operand stack. Subcomputations may be nested on the operand stack, and result in a single operand that can be used by the nesting computation.

Long integers and double-precision floating point numbers, while logically a single virtual machine operand, take two physical entries on the operand stack. Each primitive data type has specialized instructions that know how to operate on operands of that type. Operands must be operated on by operators appropriate to their type. It is illegal, for example, to push two integers and treat them as a long.

In most circumstances the top of the operand stack and the top of the Java stack are the same thing. As a result, we can simply refer to pushing or popping from the “stack”; the context and data of the operation make clear what we mean.

Garbage Collected Heap

The Java heap is the runtime data area from which class instances (objects) are allocated. The Java language is designed to be garbage collected — it does not give the programmer the ability to deallocate objects explicitly. Java does not presuppose any particular kind of garbage collection; various algorithms may be used depending on system requirements.

Java objects are always referred to and operated on indirectly, through handles. Handles may be thought of as pointers to areas allocated out of the garbage collected heap.

Method Area

The method area is analogous to the store for compiled code in conventional languages or the text segment in a UNIX process. It stores method code (compiled Java code), symbol tables, etc.

Constant Pool

Associated with each class is a constant pool. The constant pool contains the names of all fields, methods, and other such information that is used by any method in the class. At the end of the chapter containing the class file format there is a table of the constant pool types and their associated values.

When the class is first read in from memory, the class structure has two fields related to the constant pool. The `nconstants` field indicates the number of constants in this classes constant pool. The `constant_info.constants_offset` field contains an integer offset (in bytes) from the start of the class to the data which describes the constants in the class.

`constant_pool[0]` may be used by the implementation for whatever purposes it wishes.

`constant_pool[1] ... constant_pool[nconstants - 1]` are described by the sequence of bytes beginning at the byte indicated by constant_info.constants_offset in the class object. Each sequence of bytes contains a “type” field, followed by one or more type-dependent bytes, describing in more detail the specific field.

Limitations

The Java virtual machine design imposes some limitations on Java implementations based on it.

- 32-bit pointers and stacks limit the Java virtual machine’s internal addressing to 4G

- Signed 16-bit offsets (e.g. *ifeq*) for branch and jump instructions limit the size of an Java method to 32k
- Unsigned 8-bit local variable indices limit the number of local variables per Java stack frame to 256
- Signed 16-bit indices into the constant pool limit the number of constant pool entries per method to 32k

For *_quick* instructions only [See Appendix A]:

- Unsigned 8-bit offsets into objects (e.g. *invokemethod_quick*) limit the number of methods in a class to 256
- Unsigned 8-bit argument counts (e.g. *invokemethod_quick*) limits the size of a method call's parameters to 256 32 bit words, where a long or double parameters occupy two words each.

An Interpreter for the Java Instruction Set

The instruction set of the Java virtual machine can be implemented using conventional methods like compiling to native code or interpretation. Initial Java implementations will include an interpreter for the instruction set. The interpreter sees compiled Java code as a stream of bytes that it interprets as virtual machine instructions.

The inner loop of the interpreter is essentially:

```
do {
    fetch a byte
    execute an action depending on the value of the byte
    } while (there is more to do);
```

Instruction Format

Java virtual machine instructions are represented in this document by an entry of the form:

instruction name

A <u>short</u> description of the instruction

Syntax:

opcode = number
operand1
operand2
...

Stack: …., value1, value2 ⇒ ..., value3

A <u>longer description</u> that explains the functions of the instruction and indicates any exceptions that might be thrown during execution.

The items in the syntax diagram are always 8 bits wide.

The Visual Stack Representation

The effect of an instruction's execution on the operand stack is represented textually, with the stack growing from left to right. Words on the operand stack are all 32 bits wide. Thus, for

Stack: ..., *value1*, *value2* ⇒ ..., *value3*

value2 is on top of the stack with *value1* just beneath it. Both are 32-bit quantities. As a result of the execution of the instruction, *value1* and *value2* are popped from the stack and replaced by *value3*, which has been calculated by the instruction. The remainder of the stack, represented by ellipsis, is unaffected by the instruction's execution.

Long integers and double precision floats are always shown as taking up two words on the operand stack, e.g.,

Stack: ... ⇒ ..., *value*-word1, *value*-word2

Implementors are free to decide the appropriate way to divide two-word long integers and double precision floats into word1 and word2.

Conventions

Operations of the Java virtual machine most often take their operands from the stack and put their results back on the stack. As a convention, the descriptions do not usually mention when the stack is the source or destination of an operation, but will always mention when it is not. For instance, the *iload* instruction has the short description "Load integer from local variable." Implicitly, the integer is loaded onto the stack. The *iadd* instruction is described as "Integer add"; both its source and destination are the stack.

Instructions that do not affect the control flow of a computation may be assumed to always advance the virtual machine pc to the opcode of the following instruction. Only instructions that do affect control flow will explicitly mention the effect they have on pc.