The Java Virtual Machine Specification

Next

The Java Virtual Machine Specification

Table of Contents

Next

This documentation was ported to *MS Window's Help* by Bill Bercik.
Bill may be reached at: bill@dippybird.com

Stop by his web site and get the latest update to the *Information Portafilter* for Java at:
http://www.dippybird.com/java.html

Generated with CERN WebMaker

WebMaker welcome

CERN - European Laboratory for Particle Physics - PT Group

Configurable converter of FrameMaker documents to the World-Wide Web

---

The combination of WebMaker and FrameMaker enables you to publish simultaneously both the printed and the WWW versions of a document. WebMaker converts FrameMaker documents and books to a hypertext network of HTML files that may be viewed by World-Wide Web browsers such as Mosaic.

WWW is a global hypertext information network conceived at CERN, the European Laboratory for Particle Physics.

WebMaker translates FrameMaker entities such as imported and native graphics, mathematics, tables, figures, anchored frames, cross-references, character highlights, indices and footnotes. It generates tables of contents automatically, and transforms into graphical images elements that are unknown to HTML. The user has control over a number of conversion aspects:

- the rules for the breakup of the Frame document into the component HTML files;
- a panel of hypertext links to facilitate navigation within the WWW documents web;
- the rules for the mapping of paragraph and character formats to HTML constructs;
- the specification of material for selective inclusion in the FrameMaker or WWW document.

---

email: webmaker@cern.ch
Tel: +41-22-767 9393
Fax: +41-22-767 9196
URL: http://www.cern.ch/WebMaker/

Constant Pool - FootNote

FootNote

There are two differences between this format and the "standard" UTF format. First, the null byte (0x00) is encoded as two bytes rather than as one byte, so that strings never have embedded nulls. Second, only the one-byte, two-byte, and three-byte formats are used. We do not recognize the longer formats.

vmspec.: The Java Virtual Machine-Garbage Collected Heap

The Java Virtual Machine

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.

---

vmspec.: The Java Virtual Machine-Method Area

The Java Virtual Machine

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.

---

vmspec.: The Java Virtual Machine-Constant Pool

The Java Virtual Machine

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 integ er 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.

---

vmspec.: The Java Virtual Machine-Limitations

The Java Virtual Machine

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.

---

vmspec.: The Java Virtual Machine- An Interpreter for the Java Instruc

---

The Java Virtual Machine

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);
```

---

vmspec.: The Java Virtual Machine-Instruction Format

The Java Virtual Machine

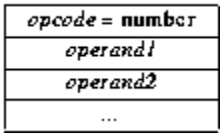
Instruction Format

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

instruction name

A short description of the instruction

Syntax:




```
xbc ., value1, value2 xde  ..., value3
```

A longer description 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

```
..., value1, value2 xde  ..., 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.,

```
... xde  ..., 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.

---

vmspec.: The Java Virtual Machine-Conventions

The Java Virtual Machine

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 al ways 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.

---

vmspec: The Virtual Machine Instruction Set

---

2 The Virtual Machine Instruction Set

---

vmspec.: The Virtual Machine Instruction Set- Pushing Constants onto t

---

The Virtual Machine Instruction Set

Pushing Constants onto the Stack

---

bipush

Push one-byte signed integer

Syntax:

bipush = 16
byte1

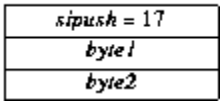
```
... =>; ..., value
```

byte1 is interpreted as a signed 8-bit value. This value is expanded to an integer and pushed onto the operand stack.

sipush

Push two-byte signed integer

Syntax:



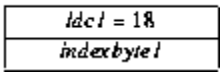

```
... =>; ..., item
```

byte1 and byte2 are assembled into a signed 16-bit value. This value is expanded to an integer and pushed onto the operand stack.

ldc1

Push item from constant pool

Syntax:



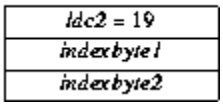

```
... =>; ..., item
```

indexbyte1 is used as an unsigned 8-bit index into the constant pool of the current class. The item at that index is resolved and pushed onto the stack.

ldc2

Push item from constant pool

Syntax:




```
... =>; ..., item
```

indexbyte1 and indexbyte2 are used to construct an unsigned 16-bit index into the constant pool of the current class. The item at that index is resolved and pushed onto the stack.

ldc2w

Push long or double from constant pool

Syntax:

ldc2w = 20
indexbyte1
indexbyte2

```
... =>; ..., constant-word1, constant-word2
```

indexbyte1 and indexbyte2 are used to construct an unsigned 16-bit index into the constant pool of the current class. The two-word constant at that index is resolved and pushed onto the stack.

aconst_null

Push null object

Syntax:

aconst_null = 1

```
... =>; ..., null
```

Push the null object onto the stack.

iconst_m1

Push integer constant -1

Syntax:

iconst_m1 = 2

```
... =>; ..., -1
```

Push the integer -1 onto the stack.

iconst_<;n>;

Push integer constant

Syntax:

iconst_<n>

<;n>;

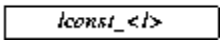
```
... =>; ..., <;n>;
```

Forms: iconst_0 = 3, iconst_1 = 4, iconst_2 = 5, iconst_3 = 6, iconst_4 = 7, iconst_5 = 8

Push the integer <;n>; onto the stack.

lconst_<;l>;

Push long integer constant
Syntax:



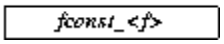

```
... =>; ..., <;l>;-word1, <;l>;-word2
```

Forms: lconst_0 = 9, lconst_1 = 10

Push the long integer <;l>; onto the stack.

fconst_<;f>;

Push single float
Syntax:



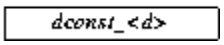

```
... =>; ..., <;f>;
```

Forms: fconst_0 = 11, fconst_1 = 12, fconst_2 = 13

Push the single precision floating point number <;f>; onto the stack.

dconst_<;d>;

Push double float
Syntax:




```
... =>; ..., <;d>;-word1, <;d>;-word2
```

Forms: dconst_0 = 14, dconst_1 = 15

Push the double precision floating point number <;d>; onto the stack.

---

vmspec.: The Virtual Machine Instruction Set- Loading Local Variables

---

The Virtual Machine Instruction Set

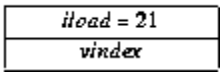
Loading Local Variables Onto the Stack

---

iload

Load integer from local variable

Syntax:



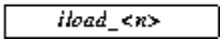

```
... =>; ..., value
```

Local variable vindex in the current Java frame should contain an integer. The value of that variable is pushed onto the operand stack.

iload_<;n>;

Load integer from local variable

Syntax:



```
... =>; ..., value
```

Forms: iload_0 = 27, iload_1 = 27, iload_2 = 28, iload_3 = 29

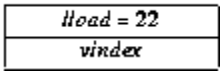
Local variable <;n>;in the current Java frame should contain an integer. The value of that variable is pushed onto the operand stack.

This instruction is the same as iload with a vindex of <;n>;, except that the operand <;n>; is implicit.

lload

Load long integer from local variable

Syntax:



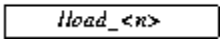

```
... =>; ..., value-word1, value-word2
```

Local variables vindex and vindex+1 in the current Java frame should together contain a long integer. The value of contained in those variables is pushed onto the operand stack.

lload_<;n>;

Load long integer from local variable

Syntax:




```
... =>; ..., value-word1, value-word2
```

Forms: lload_0 = 30, lload_1 = 31, lload_2 = 32, lload_3 = 33

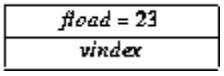
Local variables <;n>; and <;n>;+1 in the current Java frame should together contain a long integer. The value contained in those variables is pushed onto the operand stack.

This opcode is the same as lload with a vindex of <;n>;, except that the operand <;n>; is implicit.

fload

Load single float from local variable

Syntax:



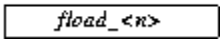

```
... =>; ..., value
```

Local variable vindex in the current Java frame should contain a single precision floating point number. The value of that variable is pushed onto the operand stack.

fload_<;n>;

Load single float from local variable

Syntax:




```
... =>; ..., value
```

Forms: fload_0 = 34, fload_1 = 35, fload_2 = 36, fload_3 = 37

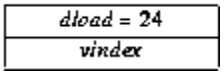
Local variable <;n>; in the current Java frame should contain a single precision floating point number. The value of that variable is pushed onto the operand stack.

This opcode is the same as fload with a vindex of <;n>;, except that the operand <;n>; is implicit.

dload

Load double float from local variable

Syntax:



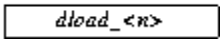

```
... =>; ..., value-word1, value-word2
```

Local variables vindex and vindex+1 in the current Java frame should together contain a double precision float point number. The value contained in those variables is pushed onto the operand stack.

dload_<;n>;

Load double float from local variable

Syntax:




```
... =>; ..., value-word1, value-word2
```

Forms: dload_0 = 38, dload_1 = 39, dload_2 = 40, dload_3 = 41

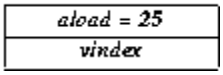
Local variables <;n>; and <;n>;+1 in the current Java frame should together contain a double precision floating point number. The value contained in those variables is pushed onto the operand stack.

This opcode is the same as dload with a vindex of <;n>;, except that the operand <;n>; is implicit.

aload

Load local object variable

Syntax:



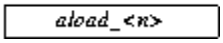

```
... =>; ..., value
```

Local variable vindex in the current Java frame should contain a handle to an object or to an array. The value of that variable is pushed onto the operand stack.

aload_<;n>;

Load object reference from local variable

Syntax:




```
... =>; ..., value
```

Forms: aload_0 = 42, aload_1 = 43, aload_2 = 44, aload_3 = 45

Local variable n in the current Java frame should contain a handle to an object or to an array. The value of that variable is pushed onto the operand stack.

This opcode is the same as aload with a vindex of <;n>;, except that the operand <;n>; is implicit.

Generated with CERN WebMaker

vmspec: The Java Virtual Machine

---

1 The Java Virtual Machine

---

vmspec.: The Virtual Machine Instruction Set- Storing Stack Values int

---

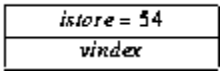
The Virtual Machine Instruction Set

Storing Stack Values into Local Variables

---

istore

Store integer into local variable
Syntax:

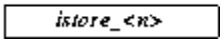



```
..., value =>; ...
```

value should be an integer. Local variable vindex in the current Java frame is set to value.

istore_<;n>;

Store integer into local variable
Syntax:



```
..., value =>; ...
```

Forms: istore_0 = 59, istore_1 = 60, istore_2 = 61, istore_3 = 62

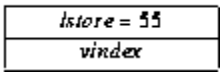
value should be an integer. Local variable <;n>; in the current Java frame is set to value.

This instruction is the same as istore with a vindex of <;n>;, except that the operand <;n>; is implicit.

lstore

Store long integer into local variable

Syntax:



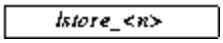

```
..., value-word1, value-word2 =>; ...
```

value should be a long integer. Local variables vindex and vindex+1 in the current Java frame are set to value.

lstore_<;n>;

Store long integer into local variable

Syntax:




```
..., value-word1, value-word2 =>; ...
```

Forms: lstore_0 = 63, lstore_1 = 64, lstore_2 = 65, lstore_3 = 66

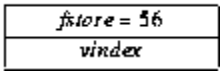
value should be a long integer. Local variables <;n>; and <;n>;+1 in the current Java frame are set to value.

This instruction is the same as lstore with a vindex of <;n>;, except that the operand <;n>; is implicit.

fstore

Store single float into local variable

Syntax:

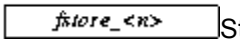



```
..., value =>; ...
```

value should be a single precision floating point number. Local variable vindex in the current Java frame is set to value.

fstore_<;n>;

Syntax:



Store single float into local variable

```
..., value =>; ...
```

Possible Instructions:

fstore_0 = 67, fstore_1 = 68, fstore_2 = 69, fstore_3 = 70

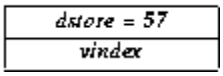
value should be a single precision floating point number. Local variable <;n>; in the current Java frame is set to value.

This instruction is the same as fstore with a vindex of <;n>;, except that the operand <;n>; is implicit.

dstore

Store double float into local variable

Syntax:

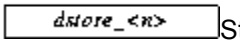



```
..., value-word1, value-word2 =>; ...
```

value should be a double precision floating point number. Local variables vindex and vindex+1 in the current Java frame are set to value.

dstore_<;n>;

Syntax:



Store double float into local variable

```
..., value-word1, value-word2 =>; ...
```

Forms: dstore_0 = 71, dstore_1 = 72, dstore_2 = 73, dstore_3 = 74

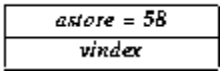
value should be an double precision floating point number. Local variables <;n>; and <;n>;+1 in the current Java frame are set to value.

This instruction is the same as dstore with a vindex of <;n>;, except that the operand <;n>; is implicit.

astore

Store object reference into local variable

Syntax:

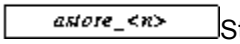



```
..., value =>; ...
```

value should be a handle to an array or to an object. Local variable vindex in the current Java frame is set to value.

astore_<;n>;

Syntax:



Store object reference into local variable

```
..., value =>; ...
```

Forms: astore_0 = 75, astore_1 = 76, astore_2 = 77, astore_3 = 78

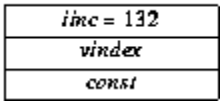
value should be a handle to an array or to an object. Local variable <;n>; in the current Java frame is set to value.

This instruction is the same as astore with a vindex of <;n>;, except that the operand <;n>; is implicit.

iinc

Increment local variable by constant

Syntax:




```
no change
```

Local variable vindex in the current Java frame should contain an integer. Its value is incremented by the value const, where const is treated as a signed 8-bit quantity.

---

vmspec.: The Virtual Machine Instruction Set- Managing Arrays

---

The Virtual Machine Instruction Set

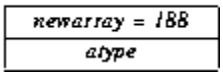
Managing Arrays

---

newarray

Allocate new array

Syntax:




```
..., size =>; result
```

size should be an integer. It represents the number of elements in the new array.

atype is an internal code that indicates the type of array to allocate. Possible values for atype are as follows:

T_ARRAY	1
T_BOOLEAN	4
T_CHAR	5
T_FLOAT	6
T_DOUBLE	7
T_BYTE	8
T_SHORT	9
T_INT	10
T_LONG	11

A new array of the indicated or computed atype, capable of holding size elements, is allocated. Allocation of an array large enough to contain nelem items of atype is attempted. All elements of the array are initialized to zero.

If size is less than zero, a NegativeArraySizeException is thrown. If there is not enough memory to allocate the array, an OutOfMemoryException is thrown.

anewarray

Allocate new array

Syntax:

anewarray = 189
indexbyte1
indexbyte2

of objects

```
..., size=>; result
```

size should be an integer. It represents the number of elements in the new array.

indexbyte1 and indexbyte2 are are used to construct an index into the constant pool of the current class. The item at that index is resolved. The resulting entry should be a class.

A new array of the indicated class type and capable of holding size elements is allocated. Allocation of an array large enough to contain size items of the given class type is attempted. All elements of the array are initialized to zero.

If size is less than zero, a NegativeArraySizeException is thrown. If there is not enough memory to allocate the array, an OutOfMemoryException is thrown.

anewarray is used to create a single dimension of an array of objects. For example, to create

```
new Thread[7]
```

the following code is used:

```
bipush 7
anewarray <;Class "java.lang.Thread">;
```

anewarray can also be used to create the outermost dimension of a multi-dimensional array. For example, the following array declaration:

```
new int[6][]
```

is created with the following code:

```
bipush 6
anewarray <;Class "[I">;
```

See CONSTANT_Class in the Class File Format chapter for information on array class names.

multianewarray

Allocate new multi-dimensional array

Syntax:

anewarray = 198
indexbyte1
indexbyte2
dimensions

```
..., size1 size2...sizen =>; result
```

Each size should be an integer. Each represents the number of elements in a dimension of the array.

indexbyte1 and indexbyte2 are are used to construct an index into the constant pool of the current class. The item at that index is resolved. The resulting entry should be a class.

dimensions has the following aspects:

- It should be an integer xb3 1.

- It represents the number of dimensions being created. It must be the number of dimensions of the array class.
- It represents the number of elements that are popped off the stack. All must be integers greater than or equal to zero. These are used as the sizes of the dimension. For example, to create:

```
new int[6][3][]
```

the following code is used:

```
bipush 6
bipush 3
multianewarray <;Class "[[[I">; 2
```

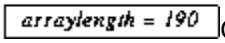
If any of the size arguments on the stack is less than zero, a NegativeArraySizeException is thrown. If there is not enough memory to allocate the array, an OutOfMemoryException is thrown.

Note: It is more efficient to use newarray or anewarray when creating a single dimension.

See CONSTANT_Class in the Class File Format chapter for information on array class names.

arraylength

Syntax:



Get length of array

```
..., handle =>; ..., length
```

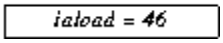
handle should be the handle of an array. The length of the array is determined and replaces handle on the top of the stack.

If the handle is null, a NullPointerException is thrown.

iaload

Load integer from array

Syntax:



```
..., array, index =>; ..., value
```

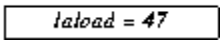
array should be an array of integers. index should be an integer. The integer value at position number index in array is retrieved and pushed onto the top of the stack.

If array is null a NullPointerException is thrown. If index is not within the bounds of array an ArrayIndexOutOfBoundsException is thrown.

laload

Load long integer from array

Syntax:




```
..., array, index =>; ..., value-word1, value-word2
```

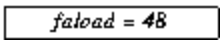
array should be an array of long integers. index should be an integer. The long integer value at position number index in array is retrieved and pushed onto the top of the stack.

If array is null a NullPointerException is thrown. If index is not within the bounds of array an ArrayIndexOutOfBoundsException is thrown.

faload

Load single float from array

Syntax:



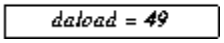

```
..., array, index =>; ..., value
```

array should be an array of single precision floating point numbers. index should be an integer. The single precision floating point number value at position number index in array is retrieved and pushed onto the top of the stack.

If array is null a NullPointerException is thrown. If index is not within the bounds of array an ArrayIndexOutOfBoundsException is thrown.

daload

Load double float from array
Syntax:



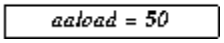

```
..., array, index =>; ..., value-word1, value-word2
```

array should be an array of double precision floating point numbers. index should be an integer. The double precision floating point number value at position number index in array is retrieved and pushed onto the top of the stack.

If array is null a NullPointerException is thrown. If index is not within the bounds of array an ArrayIndexOutOfBoundsException is thrown.

aaload

Load object reference from array
Syntax:



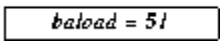

```
..., array, index =>; ..., value
```

array should be an array of handles to objects or arrays. index should be an integer. The object or array value at position number index in array is retrieved and pushed onto the top of the stack.

If array is null a NullPointerException is thrown. If index is not within the bounds of array an ArrayIndexOutOfBoundsException is thrown.

baload

Load signed byte from array
Syntax:



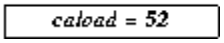

```
..., array, index =>; ..., value
```

array should be an array of signed bytes. index should be an integer. The signed byte value at position number index in array is retrieved, expanded to an integer, and pushed onto the top of the stack.

If array is null a NullPointerException is thrown. If index is not within the bounds of array an ArrayIndexOutOfBoundsException is thrown.

caload

Load character from array
Syntax:



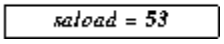

```
..., array, index =>; ..., value
```

array should be an array of characters. index should be an integer. The character value at position number index in array is retrieved, expanded to an integer, and pushed onto the top of the stack.

If array is null a NullPointerException is thrown. If index is not within the bounds of array an ArrayIndexOutOfBoundsException is thrown.

saload

Load short from array
Syntax:



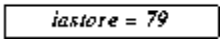

```
..., array, index =>; ..., value
```

array should be an array of (signed) short integers. index should be an integer. The short integer value at position number index in array is retrieved, expanded to an integer, and pushed onto the top of the stack.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array, an ArrayIndexOutOfBoundsException is thrown.

iastore

Store into integer array
Syntax:




```
..., array, index, value =>; ...
```

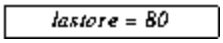
array should be an array of integers, index should be an integer, and value an integer. The integer value is stored at position index in array.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array, an

ArrayIndexOutOfBoundsException is thrown.

lastore

Store into long integer array
Syntax:



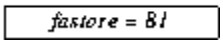

```
..., array, index, value-word1, value-word2 =>; ...
```

array should be an array of long integers, index should be an integer, and value a long integer. The long integer value is stored at position index in array.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array, an ArrayIndexOutOfBoundsException is thrown.

fastore

Store into single float array
Syntax:



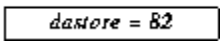

```
..., array, index, value =>; ...
```

array should be an array of single precision floating point numbers, index should be an integer, and value a single precision floating point number. The single float value is stored at position index in array.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array, an ArrayIndexOutOfBoundsException is thrown.

dastore

Store into double float array
Syntax:




```
..., array, index, value-word1, value-word2 =>; ...
```

array should be an array of double precision floating point numbers, index should be an integer, and value a double precision floating point number. The double float value is stored at position index in array.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array, an ArrayIndexOutOfBoundsException is thrown.

aastore

Store into object reference array

Syntax:

aastore = 83

```
..., array, index, value =>; ...
```

array should be an array of handles to objects or to arrays, index should be an integer, and value a handle to an object or array. The handle value is stored at position index in array.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array, an ArrayIndexOutOfBoundsException is thrown.

The actual type of value should be conformable with the actual type of the elements of the array. For example, it is legal to store and instance of class Thread in an array of class Object, but not vice versa. An IncompatibleTypeException is thrown if an attempt is made to store an incompatible object reference.

bastore

Store into signed byte array

Syntax:

bastore = 84

```
..., array, index, value =>; ...
```

array should be an array of signed bytes, index should be an integer, and value an integer. The integer value is stored at position index in array. If value is too large to be a signed byte, it is truncated.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array, an ArrayIndexOutOfBoundsException is thrown.

castore

Store into character array

Syntax:

castore = 85

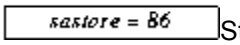
```
..., array, index, value =>; ...
```

array should be an array of characters, index should be an integer, and value an integer. The integer value is stored at position index in array. If value is too large to be a character, it is truncated.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array, an ArrayIndexOutOfBoundsException is thrown.

sastore

Syntax:



Store into short array

```
..., array, index, value =>; ...
```

array should be an array of shorts , index should be an integer, and value an integer. The integer value is stored at position index in array. If value is too large to be an short, it is truncated.

If array is null, a NullPointerException is thrown. If index is not within the bounds of array an ArrayIndexOutOfBoundsException is thrown.

---

vmspec.: The Virtual Machine Instruction Set- Stack Instructions

---

The Virtual Machine Instruction Set

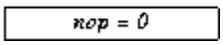
Stack Instructions

---

nop

Do nothing.

Syntax:



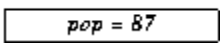

```
no change
```

Do nothing.

pop

Pop top stack word

Syntax:

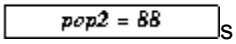



```
..., any =>; ...
```

Pop the top word from the stack.

pop2

Pop top two stack word
Syntax:



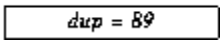
s

```
..., any2, any1 =>; ...
```

Pop the top two words from the stack.

dup

Duplicate top stack word
Syntax:

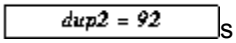



```
..., any =>; ..., any, any
```

Duplicate the top word on the stack.

dup2

Duplicate top two stack word
Syntax:



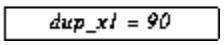
s

```
..., any2, any1 =>; ..., any2, any1, any2, any1
```

Duplicate the top two words on the stack.

dup_x1

Duplicate top stack word and put two down
Syntax:



```
..., any2, any1 =>; ..., any1, any2, any1
```

Duplicate the top word on the stack and insert the copy two words down in the stack.

dup2_x1

Duplicate top two stack words and put two down

Syntax:

dup2_x1 = 93

```
..., any3, any2, any1 =>; ..., any2,, any1, any3, any2, any1
```

Duplicate the top two words on the stack and insert the copies two words down in the stack.

dup_x2

Duplicate top stack word and put three down.

Syntax:

dup_x2 = 91

```
..., any3, any2, any1 =>; ..., any1, any3, any2, any1
```

Duplicate the top word on the stack and insert the copy three words down in the stack.

dup2_x2

Duplicate top two stack words and put three down

Syntax:

dup2_x2 = 94

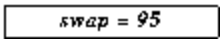
```
..., any4, any3, any2, any1 =>; ..., any2, any1, any4, any3, any2, any1
```

Duplicate the top two words on the stack and insert the copies three words down in the stack.

swap

Swap top two stack words

Syntax:




```
..., any2, any1 =>; ..., any2, any1
```

Swap the top two elements on the stack.

---

vmspec.: The Virtual Machine Instruction Set- Arithmetic Instructions

---

The Virtual Machine Instruction Set

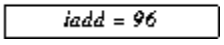
Arithmetic Instructions

---

iadd

Integer add

Syntax:




```
..., value1, value2 =>; ..., result
```

value1 and value2 should be integers. The values are added and are replaced on the stack by their integer sum.

ladd

Long integer add

Syntax:

ladd = 97

```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-word1, result-word2
```

value1 and value2 should be long integers. The values are added and are replaced on the stack by their long integer sum.

fadd

Single float add

Syntax:

fadd = 98

```
..., value1, value2 =>; ..., result
```

value1 and value2 should be single precision floating point numbers. The values are added and are replaced on the stack by their single precision floating point sum.

dadd

Double float add

Syntax:

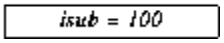
dadd = 99

```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-word1, result-word2
```

value1 and value2 should be double precision floating point numbers. The values are added and are replaced on the stack by their double precision floating point sum.

isub

Integer subtract
Syntax:

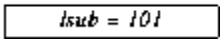



```
..., value1, value2 =>; ..., result
```

value1 and value2 should be integers. value2 is subtracted from value1, and both values are replaced on the stack by their integer difference.

lsub

Long integer subtract
Syntax:

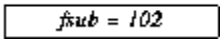



```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

value1 and value2 should be long integers. value2 is subtracted from value1, and both values are replaced on the stack by their long integer difference.

fsub

Single float subtract
Syntax:

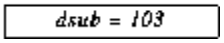



```
..., value1, value2 =>; ..., result
```

value1 and value2 should be single precision floating point numbers. value2 is subtracted from value1, and both values are replaced on the stack by their single precision floating point difference.

dsub

Double float subtract
Syntax:



```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

value1 and value2 should be double precision floating point numbers. value2 is subtracted from value1, and both values are replaced on the stack by their double precision floating point difference.

imul

Integer multiply

Syntax:

imul = 104

```
..., value1, value2 =>; ..., result
```

value1 and value2 should be integers. Both values are replaced on the stack by their integer product.

lmul

Long integer multiply

Syntax:

lmul = 105

```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

value1 and value2 should be long integers. Both values are replaced on the stack by their long integer product.

fmul

Single float multiply

Syntax:

fmul = 106

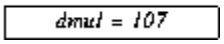
```
..., value1, value2 =>; ..., result
```

value1 and value2 should be single precision floating point numbers. Both values are replaced on the stack by their single precision floating point product.

dmul

Double float multiply

Syntax:



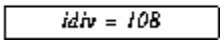

```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

value1 and value2 should be double precision floating point numbers. Both values are replaced on the stack by their double precision floating point product.

idiv

Integer divide

Syntax:




```
..., value1, value2 =>; ..., result
```

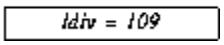
value1 and value2 should be integers. value1 is divided by value2, and both values are replaced on the stack by their integer quotient.

The result is truncated to the nearest integer that is between it and 0. An attempt to divide by zero results in a "/ by zero" ArithmeticException being thrown.

ldiv

Long integer divide

Syntax:




```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

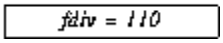
value1 and value2 should be long integers. value1 is divided by value2, and both values are replaced on the stack by their long integer quotient.

The result is truncated to the nearest integer that is between it and 0. An attempt to divide by zero results in a "/ by zero" ArithmeticException being thrown.

fdiv

Single float divide

Syntax:




```
..., value1, value2 =>; ..., result
```

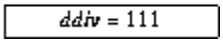
value1 and value2 should be single precision floating point numbers. value1 is divided by value2, and both values are replaced on the stack by their single precision floating point quotient.

Divide by zero results in the quotient being NaN.

ddiv

Double float divide

Syntax:




```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

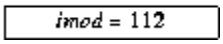
value1 and value2 should be double precision floating point numbers. value1 is divided by value2, and both values are replaced on the stack by their double precision floating point quotient.

Divide by zero results in the quotient being NaN.

imod

Integer mod

Syntax:




```
..., value1, value2 =>; ..., result
```

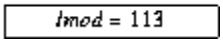
value1 and value2 should both be integers. value1 is divided by value2, and both values are replaced on the stack by their integer remainder.

An attempt to divide by zero results in a "/ by zero" ArithmeticException being thrown.

lmod

Long integer mod

Syntax:




```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-word1, result-word2
```

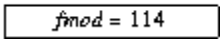
value1 and value2 should both be long integers. value1 is divided by value2, and both values are replaced on the stack by their long integer remainder.

An attempt to divide by zero results in a "/ by zero" ArithmeticException being thrown.

fmod

Single float mod

Syntax:




```
..., value1, value2 =>; ..., result
```

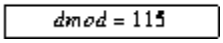
value1 and value2 should both be single precision floating point numbers. value1 is divided by value2, and the quotient is truncated to an integer, and then multiplied by value2. The product is subtracted from value1.The result, as a single precision floa ting point number, replaces both values on the stack. That is, result = value1 - ((int)(value1/value2)) * value2.

An attempt to divide by zero results in NaN.

dmod

Double float mod

Syntax:



```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

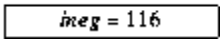
value1 and value2 should both be double precision floating point numbers. value1 is divided by value2, and the quotient is truncated to an integer, and then multiplied by value2. The product is subtracted from value1.The result, as a double precision floa ting point number, replaces both values on the stack. That is, result = value1 - ((int)(value1/value2)) * value2.

An attempt to divide by zero results in NaN.

ineg

Integer negate

Syntax:



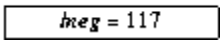

```
..., value =>; ..., result
```

value should be an integer. It is replaced on the stack by its arithmetic negation.

lneg

Long integer

Syntax:



negate

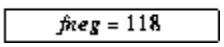
```
..., value-word1, value-word2 =>; ..., result-word1, result-word2
```

value should be a long integer. It is replaced on the stack by its arithmetic negation.

fneg

Single float negate

Syntax:



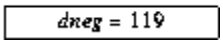

```
..., value =>; ..., result
```

value should be a single precision floating point number. It is replaced on the stack by its arithmetic negation.

dneg

Double float negate

Syntax:




```
..., value-word1, value-word2 =>; ..., result-word1, result-word2
```

value should be a double precision floating point number. It is replaced on the stack by its arithmetic negation.

---

vmspec.: The Virtual Machine Instruction Set- Logical Instructions

---

The Virtual Machine Instruction Set

Logical Instructions

---

ishl

Integer shift left

Syntax:

ishl = 120

```
..., value1, value2 =>; ..., result
```

value1 and value2 should be integers. value1 is shifted left by the amount indicated by the low five bits of value2. The integer result replaces both values on the stack.

ishr

Integer arithmetic shift right

Syntax:

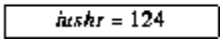
ishr = 122

```
..., value1, value2 =>; ..., result
```

value1 and value2 should be integers. value1 is shifted right arithmetically (with sign extension) by the amount indicated by the low five bits of value2. The integer result replaces both values on the stack.

iushr

Integer logical shift right
Syntax:

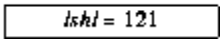



```
..., value1, value2 =>; ..., result
```

value1 and value2 should be integers. value1 is shifted right logically (with no sign extension) by the amount indicated by the low five bits of value2. The integer result replaces both values on the stack.

lshl

Long integer shift left
Syntax:

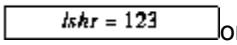



```
..., value1-word1, value1-word2, value2 =>; ..., result-word1, result-word2
```

value1 should be a long integer and value2 should be an integer. value1 is shifted left by the amount indicated by the low six bits of value2. The long integer result replaces both values on the stack.

lshr

L
Syntax:



ong integer arithmetic shift right

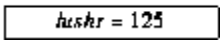
```
..., value1-word1, value1-word2, value2 =>; ..., result-word1, result-word2
```

value1 should be a long integer and value2 should be an integer. value1 is shifted right arithmetically (with sign extension) by the amount indicated by the low six bits of value2. The long integer result replaces both values on the stack.

lushr

Long integer logical shift right

Syntax:



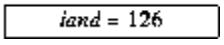

```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-word1, result-word2
```

value1 should be a long integer and value2 should be an integer. value1 is shifted right logically (with no sign extension) by the amount indicated by the low six bits of value2. The long integer result replaces both values on the stack.

iand

Integer boolean and

Syntax:



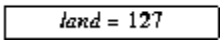

```
..., value1, value2 =>; ..., result
```

value1 and value2 should both be integers. They are replaced on the stack by their bitwise conjunction (AND).

land

Long integer boolean and

Syntax:



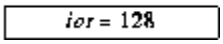

```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-word1, result-word2
```

value1 and value2 should both be long integers. They are replaced on the stack by their bitwise conjunction (AND).

ior

Integer boolean or

Syntax:



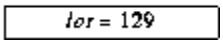

```
..., value1, value2 =>; ..., result
```

value1 and value2 should both be integers. They are replaced on the stack by their bitwise disjunction (OR).

lor

Long integer boolean or

Syntax:



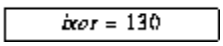

```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

value1 and value2 should both be long integers. They are replaced on the stack by their bitwise disjunction (OR).

ixor

Integer boolean xor

Syntax:



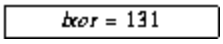

```
..., value1, value2 =>; ..., result
```

value1 and value2 should both be integers. They are replaced on the stack by their bitwise exclusive disjunction (XOR).

lxor

Long integer boolean xor

Syntax:




```
..., value1-word1, value1-word2, value2-word1, value2-word2 =>; ..., result-
word1, result-word2
```

value1 and value2 should both be long integers. They are replaced on the stack by their bitwise exclusive disjunction (XOR).

---

vmspec.: The Virtual Machine Instruction Set- Conversion Operations

---

The Virtual Machine Instruction Set

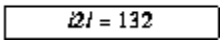
Conversion Operations

---

i2l

Integer to long integer conversion

Syntax:



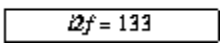

```
..., value =>; ..., result-word1, result-word2
```

value should be an integer. It is converted to a long integer. The result replaces value on the stack.

i2f

Integer to single float

Syntax:



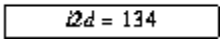
```
..., value =>; ..., result
```

value should be an integer. It is converted to a single precision floating point number. The result replaces value on the stack.

i2d

Integer to double float

Syntax:



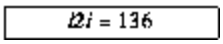

```
..., value =>; ..., result-word1, result-word2
```

value should be an integer. It is converted to a double precision floating point number. The result replaces value on the stack.

l2i

Long integer to integer

Syntax:



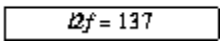

```
..., value-word1, value-word2 =>; ..., result
```

value should be a long integer. It is converted to a integer. The result replaces value on the stack.

l2f

Long integer to single float

Syntax:



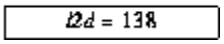

```
..., value-word1, value-word2 =>; ..., result
```

value should be a long integer. It is converted to a single precision floating point number. The result replaces value on the stack.

l2d

Long integer to double float

Syntax:



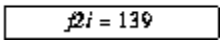

```
..., value-word1, value-word2 =>; ..., result-word1, result-word2
```

value should be a long integer. It is converted to a double precision floating point number. The result replaces value on the stack.

f2i

Single float to integer

Syntax:



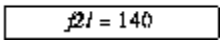

```
..., value =>; ..., result
```

value should be a single precision floating point number. It is converted to an integer. The result replaces value on the stack.

f2l

Single float to long integer

Syntax:




```
..., value =>; ..., result-word1, result-word2
```

value should be a single precision floating point number. It is converted to a long integer. The result replaces value on the stack.

f2d

Single float to double float
Syntax:

f2d = 141

```
..., value =>; ..., result-word1, result-word2
```

value should be a single precision floating point number. It is converted to a double precision floating point number. The result replaces value on the stack.

d2i

Double float to integer
Syntax:

d2i = 142

```
..., value-word1, value-word2 =>; ..., result
```

value should be a double precision floating point number. It is converted to an integer. The result replaces value on the stack.

d2l

Double float to long integer
Syntax:

d2l = 143

```
..., value-word1, value-word2 =>; ..., result-word1, result-word2
```

value should be a double precision floating point number. It is converted to a long integer. The result replaces value on the stack.

d2f

Double float to single float
Syntax:

d2f = 144

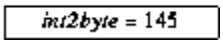
```
..., value-word1, value-word2 =>; ..., result
```

value should be a double precision floating point number. It is converted to a single precision floating point number. The result replaces value on the stack.

int2byte

Integer to signed byte

Syntax:

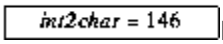



```
..., value =>; ..., result-word1, result-word2
```

value should be an integer. It is truncated to a signed 8-bit result, then sign extended to an integer. The result replaces value on the stack.

int2char

Syntax:



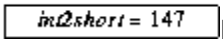
Integer to char

```
..., <;int>; =>; ..., <;result>;
```

value should be an integer. It is truncated to an unsigned 16-bit result, then sign extended to an integer. The result replaces value on the stack.

int2short

Syntax:



Integer to char

```
..., <;int>; =>; ..., <;result>;
```

value should be an integer. It is truncated to a signed 16-bit result, then sign extended to an integer. The result replaces value on the stack.

Generated with CERN WebMaker

vmspec.: The Virtual Machine Instruction Set- Control Transfer Instruc

---

The Virtual Machine Instruction Set

Control Transfer Instructions

---

ifeq

Branch if equal

Syntax:

ifeq = 153
branchbyte1
branchbyte2

to 0

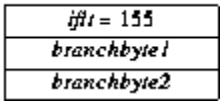
```
..., value =>; ...
```

value should be an integer or a handle to an object or to an array. It is popped from the stack. If value is equal to zero, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the ifeq.

iflt

Branch if less than

Syntax:



0

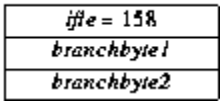
```
..., value =>; ...
```

value should be an integer. It is popped from the stack. If value is less than zero, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the iflt.

ifle

Branch if less than or equal

Syntax:



to 0

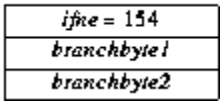
```
..., value =>; ...
```

value should be an integer. It is popped from the stack. If value is less than or equal to zero, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the ifle.

ifne

Branch if not equal

Syntax:



to 0

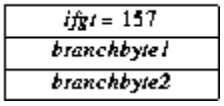
```
..., value =>; ...
```

value should be an integer or a handle to an object or to an array. It is popped from the stack. If value is not equal to zero, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the ifne.

ifgt

Branch if greater than

Syntax:



0

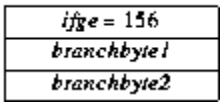
```
..., value =>; ...
```

value should be an integer. It is popped from the stack. If value is greater than zero, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the ifgt.

ifge

Branch if greater than or equal

Syntax:



to 0

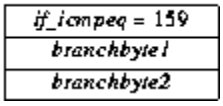
```
..., value =>; ...
```

value should be an integer. It is popped from the stack. If value is greater than or equal to zero, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the ifge.

if_icmpeq

Branch if integers equal

Syntax:



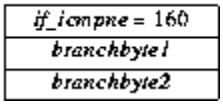

```
..., value1, value2 =>; ...
```

value1 and value2 should be integers. They are both popped from the stack. If value1 is equal to value2, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the if_icmpeq.

if_icmpne

Branch if integers not equal

Syntax:



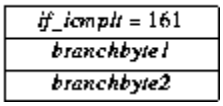

```
..., value1, value2 =>; ...
```

value1 and value2 should be integers. They are both popped from the stack. If value1 is not equal to value2, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the if_icmpne.

if_icmplt

Branch if integer less than

Syntax:




```
..., value1, value2 =>; ...
```

value1 and value2 should be integers. They are both popped from the stack. If value1 is less than value2, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the if_icmplt.

if_icmpgt

Branch if integer greater than
Syntax:

if_icmpgt = 163
branchbyte1
branchbyte2

```
..., value1, value2 =>; ...
```

value1 and value2 should be integers. They are both popped from the stack. If value1 is greater than value2 (C's >;), branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the if_icmpgt.

if_icmple

Branch if integer less than or equal to
Syntax:

if_icmple = 164
branchbyte1
branchbyte2

```
..., value1, value2 =>; ...
```

value1 and value2 should be integers. They are both popped from the stack. If value1 is less than or equal to value2, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the if_icmple.

if_icmpge

Branch if integer greater than or equal to
Syntax:

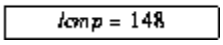
if_icmpge = 162
branchbyte1
branchbyte2

```
..., value1, value2 =>; ...
```

value1 and value2 should be integers. They are both popped from the stack. If value1 is greater than or equal to value2, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the if_icmpge.

lcmp

Long integer compare
Syntax:

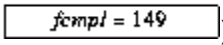



```
..., value1-word1, value1-word2, value2-word1, value2-word1 =>; ..., result
```

value1 and value2 should be long integers. They are both popped from the stack and compared. If value1 is greater than value2, the integer value 1 is pushed onto the stack. If value1 is equal to value2, the value 0 is pushed onto the stack. If value1 is less than value2, the value -1 is pushed onto the stack.

fcmpl

Single float compare (-1 on incomparable
Syntax:



)

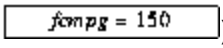
```
..., value1, value2 =>; ..., result
```

value1 and value2 should be single precision floating point numbers. They are both popped from the stack and compared. If value1 is greater than value2, the integer value 1 is pushed onto the stack. If value1 is equal to value2, the value 0 is pushed onto the stack. If value1 is less than value2, the value -1 is pushed onto the stack.

If either value1 or value2 is NaN, the value -1 is pushed onto the stack.

fcmpg

Single float compare (1 on incomparable
Syntax:



)

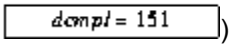
```
..., value1, value2 =>; ..., result
```

value1 and value2 should be single precision floating point numbers. They are both popped from the stack and compared. If value1 is greater than value2, the integer value 1 is pushed onto the stack. If value1 is equal to value2, the value 0 is pushed onto the stack. If value1 is less than value2, the value -1 is pushed onto the stack.

If either value1 or value2 is NaN, the value 1 is pushed onto the stack.

dcmpl

Double float compare (-1 on incomparable
Syntax:



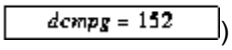

```
..., value1-word1, value1-word2, value2-word1, value2-word1 =>; ..., result
```

value1 and value2 should be double precision floating point numbers. They are both popped from the stack and compared. If value1 is greater than value2, the integer value 1 is pushed onto the stack. If value1 is equal to value2, the value 0 is pushed onto the stack. If value1 is less than value2, the value -1 is pushed onto the stack.

If either value1 or value2 is NaN, the value -1 is pushed onto the stack.

dcmpg

Double float compare (1 on incomparable
Syntax:




```
..., value1-word1, value1-word2, value2-word1, value2-word1 =>; ..., result
```

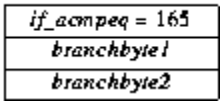
value1 and value2 should be double precision floating point numbers. They are both popped from the stack and compared. If value1 is greater than value2, the integer value 1 is pushed onto the stack. If value1 is equal to value2, the value 0 is pushed onto the stack. If value1 is less than value2, the value -1 is pushed onto the stack.

If either value1 or value2 is NaN, the value 1 is pushed onto the stack.

if_acmpeq

Branch if objects same

Syntax:



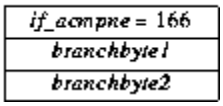

```
..., value1, value2 =>; ...
```

value1 and value2 should be handles to objects or arrays. They are both popped from the stack. If value1 is equal to value2, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the if_acmpeq.

if_acmpne

Branch if objects not same

Syntax:



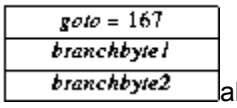

```
..., value1, value2 =>; ...
```

value1 and value2 should be handles to objects or arrays. They are both popped from the stack. If value1 is not equal to value2, branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc. Otherwise execution proceeds at the instruction following the if_acmpne.

goto

Branch

Syntax:



always

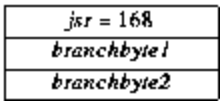
```
no change
```

branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. Execution proceeds at that offset from the pc.

jsr

Jump subroutine

Syntax:




```
... =>; ..., return-address
```

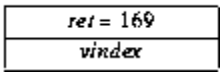
branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset. The address of the instruction immediately following the jsr is pushed onto the stack. Execution proceeds at the offset from the current pc.

The jsr instruction is used in the implementation of Java's finally keyword.

ret

Return from subroutine

Syntax:




```
no change
```

Local variable vindex in the current Java frame should contain a return address. The contents of the local variable are written into the pc.

Note that jsr pushes the address onto the stack, and ret gets it out of a local variable. This asymmetry is intentional.

The ret instruction is used in the implementation of Java's finally keyword.

---

vmspec.: The Virtual Machine Instruction Set- Function Return

---

The Virtual Machine Instruction Set

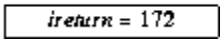
Function Return

---

ireturn

Return integer from function

Syntax:




```
..., value =>; [empty]
```

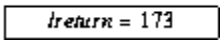
value should be an integer. The value value is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

[Note: this may be confusing to people expecting that the stack is like the C stack. However, the operand stack should be seen as consisting of a number of discontiguous segments, each corresponding to a method invocation. A return instruction empties the Java operand stack segment corresponding to the activity of the returning invocation, but does not affect the segment of any parent invocations.]]

lreturn

Return long integer from function

Syntax:



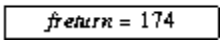
```
..., value-word1, value-word2 =>; [empty]
```

value should be a long integer. The value value is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

freturn

Return single float from function

Syntax:



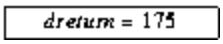

```
..., value =>; [empty]
```

value should be a single precision floating point number. The value value is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

dreturn

Return double float from function

Syntax:



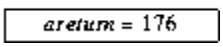

```
..., value-word1, value-word2 =>; [empty]
```

value should be a double precision floating point number. The value value is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

areturn

Return object reference from function

Syntax:



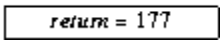

```
..., value =>; [empty]
```

value should be a handle to an object or an array. The value value is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

return

Return (void) from procedure

Syntax:




```
... =>; [empty]
```

All values on the operand stack are discarded. The interpreter then returns control to its caller.

---

vmspec.: The Virtual Machine Instruction Set- Table Jumping

The Virtual Machine Instruction Set

Table Jumping

---

tableswitch

Access jump table by index and jump

Syntax:

tableswitch = 170
...0-3 byte pad...
default-offset1
default-offset2
default-offset3
default-offset4
low1
low2
low3
low4
high1
high2
high3
high4
...jump offsets...

```
..., index =>; ...
```

tableswitch is a variable length instruction. Immediately after the tableswitch opcode, between zero and three 0's are inserted as padding so that the next byte begins at an address that is a multiple of four. After the padding follow a series of signed 4 -byte quantities: default-offset, low, high, and then high-low+1 further signed 4-byte offsets. The high-low+1 signed 4-byte offsets are treated as a 0-based jump table.

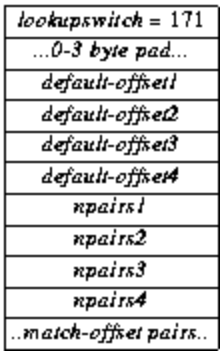
The index should be an integer. If index is less than low or index is greater than high, then default-offset is added to the pc. Otherwise, low is subtracted from index, and the index-low'th element of the jump

table is extracted, and added to the pc.

lookupswitch

Access jump table by key match and jump

Syntax:




```
..., key =>; ...
```

lookupswitch is a variable length instruction. Immediately after the lookupswitch opcode, between zero and three 0's are inserted as padding so that the next byte begins at an address that is a multiple of four.

Immediately after the padding are a series of pairs of signed 4-byte quantities. The first pair is special. The first item of that pair is the default offset, and the second item of that pair gives the number of pairs that follow. Each subsequent pair consists of a match and an offset.

The key should be an integer. The integer key on the stack is compared against each of the matches. If it is equal to one of them, the offset is added to the pc. If the key does not match any of the matches, the default offset is added to the pc.

---

vmspec.: The Virtual Machine Instruction Set- Manipulating Object Fiel

---

The Virtual Machine Instruction Set

Manipulating Object Fields

---

putfield

Set field in object

Syntax:

putfield = 181
indexbyte1
indexbyte2

```
..., handle, value =>; ...
```

OR

```
..., handle, value-word1, value-word2 =>; ...
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a class name and a field name. The item is resolved to a field block pointer which has both the field width (in bytes) and the field offset (in bytes).

The field at that offset from the start of the instance pointed to by handle will be set to the value on the top of the stack.

This instruction handles both 32-bit and 64-bit wide fields.

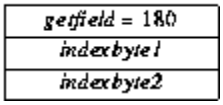
If handle is null, a NullPointerException exception is generated.

If the specified field is a static field, a DynamicRefOfStaticField exception is generated.

getfield

Fetch field from object

Syntax:




```
..., handle =>; ..., value
```

OR

```
..., handle =>; ..., value-word1, value-word2
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a class name and a field name. The item is resolved to a field block pointer which has both the field width (in bytes) and the field offset (in bytes).

handle should be a handle to an object. The value at offset into the object referenced by handle replaces handle on the top of the stack.

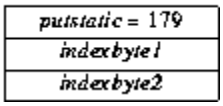
This instruction handles both 32-bit and 64-bit wide fields.

If the specified field is a static field, a DynamicRefOfStaticField exception is generated.

putstatic

Set static field in class

Syntax:




```
..., value =>; ...
```

OR

```
..., value-word1, value-word2 =>; ...
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. That field will be set to have the value on the top of the stack.

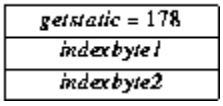
This instruction works for both 32-bit and 64-bit wide fields.

If the specified field is a dynamic field, a StaticRefOfDynamicFieldException is generated.

getstatic

Get static field from class

Syntax:




```
..., =>; ..., value
```

OR

```
..., =>; ..., value-word1, value-word2
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The value of that field will replace handle on the stack.

This instruction handles both 32-bit and 64-bit wide fields.

If the specified field is a dynamic field, a StaticRefOfDynamicFieldException is generated.

---

vmspec.: The Java Virtual Machine-About the Spec

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 WebRunner/Java 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 de tails 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 WebRunner/Java 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 WebRunner 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 logica l 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 pl ay 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.

---

vmspec.: The Virtual Machine Instruction Set- Method Invocation

---

The Virtual Machine Instruction Set

Method Invocation

There are four instructions that implement different flavors of method invocation. At first glance their descriptions look very similar but they are all slightly different.

invokevirtual
: Searches for a non-static method through an object instance, taking into account the runtime type of the object being referenced. It's behavior is similar to that of virtual methods in C++.

invokenonvirtual
: Searches for a non-static method beginning in a particular class. Behaves like non-virtual methods in C++.

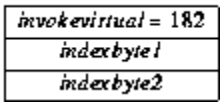
invokestatic
: Searches for a static method in a particular class.

invokeinterface
: Begins searching with the most derived class of the object, like invokemethod, but it does not presume to know which slot the method will be found in. It's behavior is similar to mutiply-inherited virtual methods in C++.

invokevirtual

Invoke class method

Syntax:




```
..., object, [arg1, [arg2 ...]], ... =>; ...
```

The operand stack is assumed to contain a handle to an object or to an array and some number of arguments. indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item at that index in the constant pool co ntains the complete method signature. A pointer to the object's method table is retrieved from the object handle. The method signature is looked up in the the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is an index into the method table of the named class, where a pointer to the method block for the matched method is found. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (nargs) expected on the operand stack.

If the method is marked synchronized the monitor associated with handle is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

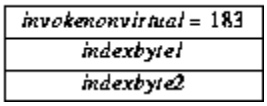
The base of the local variables array for the new Java stack frame is set to point to handle on the stack, making handle and the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables use d by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a NullPointerException is thrown. If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

invokenonvirtual

Invoke non-virtual method

Syntax:




```
..., object, nargs, ... =>; ...
```

The operand stack is assumed to contain a handle to an object and some number of arguments. indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the comp lete method signature. A pointer to the object's method table is retrieved from the object handle. The method signature is looked up in the the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (nargs) expected on the operand stack.

If the method is marked synchronized the monitor associated with handle is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to handle on the stack, making handle and the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables use d by the method is determined, and the execution

environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a NullPointerException is thrown. If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

invokestatic

Invoke a static method

Syntax:

invokestatic = 184
indexbyte1
indexbyte2

```
..., , nargs, ... =>; ...
```

The operand stack is assumed to contain some number of arguments. indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature and class. The method signature is looked up in the the method table of the class indicated. The method signature is guaranteed to exactly match one of the method signatures in the class's method table.

The result of the lookup is a method block. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (nargs) expected on the operand stack.

If the method is marked synchronized the monitor associated with the class is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to the first argument on the stack, making the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables us ed by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

invokeinterface

Invoke interface method

Syntax:

invokeinterface = 185
indexbyte1
indexbyte2
nargs
reserved

```
..., object, [arg1, [arg2 ...]], ... =>; ...
```

The operand stack is assumed to contain a handle to an object and nargs-1 arguments. indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete me thod signature. A pointer to the object's method table is retrieved from the object handle. The method signature is looked up in the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (native, synchronized, etc.) but unlike invokemethod and invokesuper, the number of available arguments (nargs) is taken from the bytecode.

If the method is marked synchronized the monitor associated with handle is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to handle on the stack, making handle and the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables use d by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a NullPointerException is thrown. If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

---

vmspec.: The Virtual Machine Instruction Set- Exception Handling

---

The Virtual Machine Instruction Set

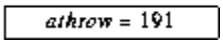
Exception Handling

The virtual machine support for exceptions documented here is likely to change in the near future but reflects the current Java implementation. The instructions here also assume that asynchronous exceptions are not supported.
athrow

The virtual machine support for exceptions documented here is likely to change in the near future but reflects the current Java implementation. The instructions here also assume that asynchronous exceptions are not supported.

athrow

Throw exception
Syntax:




```
..., handle =>; [undefined]
```

handle should be a handle to an object. The handle should be of an exception object, which is thrown. The current Java stack frame is searched for the most recent catch clause that handles this exception. A catch clause can handle an exception if the obje ct in the constant pool at for that entry is a superclass of the thrown object.) If a matching catch list entry is found, the pc is reset to the address indicated by the catch-list pointer, and execution continues there.

If no appropriate catch clause is found in the current stack frame, that frame is popped and the exception is rethrown. If one is found, it contains the location of the code for this exception. The pc is reset to that location and execution continues. If no appropriate catch is found in the current stack frame, that frame is popped and the exception is rethrown.

If handle is null, then a NullPointerException is thrown instead.

vmspec.: The Virtual Machine Instruction Set- Miscellaneous Object Ope

---

The Virtual Machine Instruction Set

Miscellaneous Object Operations

---

new

Create new object
Syntax:

new = 187
indexbyte1
indexbyte2

```
... =>; ..., handle
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item at that index should be a class name that can be resolved to a class pointer, class. A new instance of that class is then created and a handle for it is pushed on the stack.

newfromname

Create new object
Syntax:

newfromname = 186

from name

```
..., handle =>; ..., new-handle
```

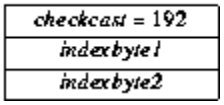
handle should be a handle to a character array. The class whose name is the string represented by the character array is determined. A new object of that class is created, and a handle new-handle for that object replaces the character array handle on the top of the stack.

If the handle is null, a NullPointerException is thrown. If no such class can be found, a NoClassDefFoundException is thrown.

checkcast

Make sure object is of given type

Syntax:




```
..., handle =>; ..., [handle|...]
```

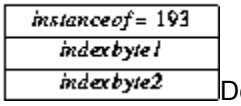
indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The string at that index of the constant pool is presumed to be a class name which can be resolved to a class pointer, class. handle should be a handle to an object.

checkcast determines whether handle can be cast to an object of class class. A null handle can be cast to any class. Otherwise handle must be an instance of class or one of its superclasses. If handle can be cast to class execution proceeds at the next instruction, and the handle for handle remains on the stack.

If handle cannot be cast to class, a ClassCastException is thrown.

instanceof

Syntax:



Determine if object is of given type

```
..., handle =>; ..., result
```

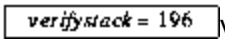
indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The string at that index of the constant pool is presumed to be a class name which can be resolved to a class pointer, class. handle should be a handle to an object.

instanceof determines whether handle can be cast to an object of the class class. This instruction will

overwrite handle with 1 if handle is null or if it is an instance of class or one of its superclasses. Otherwise, handle is overwritten by 0.

verifystack

Syntax:



Verify stack empty

```
... =>; [empty stack]
```

This instruction is only generated if the code was compiled using a debugging version of the compiler. This instruction indicates that the compiler expects the operand stack to be empty at this point.

If the stack is not currently empty, it will be set to empty. In addition, if running a debugging version of the interpreter, an error message is printed out warning that something is seriously wrong.

---

vmspec.: The Virtual Machine Instruction Set- Monitors

The Virtual Machine Instruction Set

Monitors

---

monitorenter

Syntax:

monitorenter = 194 Enter monitored region of code

```
..., handle =>; ...
```

handle should be a handle to an object.

The interpreter attempts to obtain exclusive access via a lock mechanism to handle. If another process already has handle locked, than the current process waits until the handle is unlocked. If the current process already has handle locked, then continue execution. If handle has no lock on it, then obtain an exclusive lock.

monitorexit

Syntax:

monitorexit = 195 Exit monitored region of code

```
..., handle =>; ...
```

handle should be a handle to an object.

The lock on handle is released. If this is the last lock that this process has on that handle (one process is

allowed to have multiple locks on a single handle), then other processes that are waiting for handle to be free are allowed to proceed.

---

vmspec.: The Virtual Machine Instruction Set- Debugging

The Virtual Machine Instruction Set

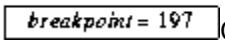
Debugging

breakpoint

---

breakpoint

Syntax:



Call breakpoint handler

The breakpoint instruction is used to temporarily overwrite an instruction causing a break to the debugger prior to the effect of the overwritten instruction. The original instruction's operands (if any) are not overwritten, and the original instruction can be restored when the breakpoint instruction is removed.

---

vmspec: Class File Format

---

3 Class File Format

---

vmspec.: Class File Format- Important Note

Class File Format

Important Note

This chapter documents the Java class file format. An important objective of Java as used in WebRunner is that alternative implementations of Java can exist and interact by sharing class files. For this to be possible, these Java implementations must pre cisely implement the design given here. Elements of the design not covered by this document are not crucial to class file sharing and may be implemented as you choose.

Please contact us directly with any questions about which design elements are essential to a modified or original Java implementation, or for help validating an Java implementation.

---

vmspec.: Class File Format- Overview

Class File Format

Overview

Class files are used to hold compiled versions of both Java classes and Java Interfaces. Compliant Java interpreters must be capable of dealing with all class files that conform to the following specification.

An Java .class file consists of a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two or four 8-bit bytes, respectively. The bytes are joined together in big-endian order.

The class file format is described in terms similar to a C structure. However, unlike a C structure,

- There is no "padding" or "alignment" between pieces of the structure.
- Each field of the structure may be of variable size.
- An array may be of variable size. In this case, some field prior to the array will give the array's dimension.

We use the types u1, u2, and u4 to mean an unsigned one-, two-, or four-byte quantity, respectively.

Attributes are used at several different places in the class format. All attributes have the following format:

```
GenericAttribute_info {

        u2      attribute_name;

        u4      attribute_length;

        u1      info[attribute_length];

}
```

The attribute_name is a 16-bit index into the class's constant pool; the value of constant_pool[attribute_name] will be a string giving the name of the attribute. The field attribute_length gives the length of the subsequent information in bytes. This length does not include the four bytes of the attribute_name and attribute_length.

In the following text, whenever we allow attributes, we give the name of the attributes that are currently understood. In the future, more attributes will be added. Class file readers are expected to skip over and ignore the information in any attributes that they do not understand.

---

vmspec.: Class File Format- Format

Class File Format

Format

The following pseudo-structure gives a top-level description of the format of a class file:

```
ClassFile {

        u4      magic;
```

```
        u4      version;

        u2      constant_pool_count;

        cp_info constant_pool[constant_pool_count - 1];

        u2      access_flags;

        u2      this_class;

        u2      super_class;

        u2      interfaces_count;

        u2      interfaces[interfaces_count];

        u2      fields_count;

        field_info      fields[fields_count];

        u2      methods_count;

        method_info     methods[methods_count];

        u2      attributes_count;

        attribute_info  attributes[attribute_count];

}
```

magic

This field must have the value 0xCAFEBABE.

version

This field contains the version number of the Java compiler that produced this class file. Different version numbers indicate incompatible changes to either the format of the class file or to the bytecodes.

The current Java version number is 45.

constant_pool_count

This field indicates the number of entries in the constant pool table.

constant_pool

The constant pool is an array of values. These values are the various string constants, class names, field names, and others that are referred to by the class structure or by the code.

constant_pool[0] is always unused. The values of constant_pool entries 1 through constant_pool_count-1 are described by the bytes that follow. These bytes are explained more fully in the section "The Constant Pool."

access_flags

This field is a set of sixteen flags used by classes, methods, and fields to describe various properties of the field, method, or class. The flags are also used to show how they can be accessed by methods in other classes. Below is a table of all the access flags. The flags that are used by classes are ACC_PUBLIC, ACC_FINAL, and ACC_INTERFACE.

Flag Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Visible to everyone	Class, Method, Variable
ACC_PRIVATE	0x0002	Visible only to the defining class	Method, Variable
ACC_PROTECTED	0x0004	Visible to subclasses	Method, Variable
ACC_STATIC	0x0008	Variable or method is static	Method, Variable
ACC_FINAL	0x0010	No further subclassing, overriding	Class, Method, Variable
ACC_SYNCHRONIZED	0x0020	Wrap method call in monitor lock	Method
ACC_THREADSAFE	0x0040	Can cache in registers	Variable
ACC_TRANSIENT	0x0080	Not written or read by the persistent object manager	Variable
ACC_NATIVE	0x0100	Implemented in C	Method
ACC_INTERFACE	0x0200	Is an interface	Class
ACC_ABSTRACT	0x0400	No definition provided	Method

Access Flags

this_class

This value is an index into the constant pool. constant_pool[this_class] must be a class, and gives the index of this class in the constant pool.

super_class

This value is an index into the constant pool. If the value of super_class is non-zero, then constant_pool[super_class] must be a class, and gives the index of this class's superclass in the constant pool.

If the value of super_class is zero, then the class being defined must be Object, and it has no superclass.

interfaces_count

This field gives the number of interfaces that this class implements.

interfaces

Each value in the array is an index into the constant pool. If an array value is non-zero, then constant_pool[interfaces[i]], for 0 <;= i <; interfaces_count, must be a class, and gives the index of an interface that this class implements.

fields_count

This value gives the number of instance variables, both static and dynamic, defined by this class. This array only includes those variables that are defined explicitly by this class. It does not include those instance variables that are accessible from this class but are inherited from super classes.

fields

Each value is a more complete description of a field in the class. See the section "Fields" for more information on the field_info structure.

methods_count

This value gives the number of methods, both static and dynamic, defined by this class. This array only includes those methods that are explicitly defined by this class. It does not include inherited methods.

methods

Each value is a more complete description of a method in the class. See the section "Methods" for more information on the method_info structure.

attributes_count

This value gives the number of additional attributes about this class.

attributes

A class can have any number of optional attributes associated with it. Currently, the only class attribute recognized is the "SourceFile" attribute, which gives the name of the source file from which this class file was compiled.

Source File Attribute

The "SourceFile" attribute has the following format:

```
SourceFile_attribute {

        u2      attribute_name_index;

        u2      attribute_length;

        u2      sourcefile_index;

}
```

attribute_name_index

constant_pool[attribute_name_index] is the string "SourceFile."

attribute_length

The length of a SourceFile_attribute must be 2.

sourcefile_index

constant_pool[sourcefile_index] is a string giving the source file from which this class file was compiled.

Fields

The information for each field immediately follows the field_count field in the class file. Each field is described by a variable length field_info structure. The format of this structure is as follows:

```
field_info {

        u2      access_flags;
```

```
        u2      name_index;

        u2      signature_index;

        u2      attributes_count;

        attribute_info  attributes[attribute_count];

}
```

access_flags

This is a set of sixteen flags used by classes, methods, and fields to describe various properties and how they many be accessed by methods in other classes. See the table "Access Flags" on page 53 which gives the meaning of the bits in this field.

The possible fields that can be set for a field are ACC_PUBLIC, ACC_PRIVATE, ACC_PROTECTED, ACC_STATIC, ACC_FINAL, ACC_THREADSAFE, and ACC_TRANSIENT.

At most one of ACC_PUBLIC and ACC_PRIVATE can be set for any method.

name_index

constant_pool[name_index] is a string which is the name of the field.

signature_index

constant_pool[signature_index] is a string which is the signature of the field. See the section "Signatures" for more information on signatures.

attributes_count

This value gives the number of additional attributes about this field.

attributes

A field can have any number of optional attributes associated with it. Currently, the only field attribute recognized is the "ConstantValue" attribute, which indicates that this field is a static numeric constant, and gives the constant value of that field.

Any other attributes are skipped.

Constant Value Attribute

The "ConstantValue" attribute has the following format:

```
ConstantValue_attribute {

        u2      attribute_name_index;

        u2      attribute_length;

        u2      constantvalue_index;

}
```

attribute_name_index

constant_pool[attribute_name_index] is the string "SourceFile."

attribute_length

The length of a SourceFile_attribute must be 2.

constantvalue_index

constant_pool[constantvalue_index]gives the constant value for this field.

The constant pool entry must be of a type appropriate to the field, as shown by the following table:

long	CONSTANT_Long
float	CONSTANT_Float
double	CONSTANT_Double
int, short, char, byte, boolean	CONSTANT_Integer

---

vmspec.: Class File Format- Methods

Class File Format

Methods

The information for each method immediately follows the method_count field in the class file. Each method is described by a variable length method_info structure. The structure has the following format:

```
method_info {

        u2      access_flags;

        u2      name_index;

        u2      signature_index;

        u2      attributes_count;

        attribute_info  attributes[attribute_count];

}
```

access_flags

This is a set of sixteen flags used by classes, methods, and fields to describe various properties and how they many be accessed by methods in other classes. See the table "Access Flags" on page 53 which gives the various bits in this field.

The possible fields that can be set for a method are ACC_PUBLIC, ACC_PRIVATE, ACC_PROTECTED, ACC_STATIC, ACC_FINAL, ACC_SYNCHRONIZED, ACC_NATIVE, and ACC_ABSTRACT.

At most one of ACC_PUBLIC and ACC_PRIVATE can be set for any method.

name_index

constant_pool[name_index] is a string giving the name of the method.

signature_index

constant_pool[signature_index]is a string giving the signature of the field. See the section "Signatures" for more information on signatures.

attributes_count

This value gives the number of additional attributes about this field.

attributes

A field can have any number of optional attributes associated with it. Each attribute has a name, and other additional information. Currently, the only field attribute recognized is the "Code" attribute, which

describes the virtual bytecode that can be executed to perform this method.

Any other attributes are skipped.

Code Attribute

The "Code" attribute has the following format:

```
Code_attribute {

        u2      attribute_name_index;

        u2      attribute_length;

        u1      max_stack;

        u1      max_locals;

        u2      code_length;

        u1      code[code_length];

        u2      exception_table_length;

        {  u2      start_pc;

           u2      end_pc;

           u2      handler_pc;

           u2      catch_type;

        }       exception_table[exception_table_length];

        u2      attributes_count;

        attribute_info  attributes[attribute_count];
```

```
}
```

attribute_name_index

constant_pool[attribute_name_index] is the string "Code."

attribute_length

This field gives the total length of the "Code" attribute, excluding the initial four bytes.

max_stack

Maximum number of entries on the operand stack that will be used during execution of this method. See the other chapters in this spec for more information on the operand stack.

max_locals

Number of local variable slots used by this method. See the other chapters in this spec for more information on the local variables.

code_length

The number of bytes in the virtual machine code for this method.

code

These are the actual bytes of the virtual machine code that implement the method. When read into memory, the first byte of code must be aligned onto a multiple-of-four boundary. See the definitions of the the opcodes "tableswitch" and "tablelookup" for more information on alignment requirements.

exception_table_length

The number of entries in the following exception table.

exception_table

Each entry in the exception table describes one exception handler in the code.

start_pc, end_pc

The two fields start_pc and end_pc give the ranges in the code at which the exception handler is active. The values of both fields are offsets from the start of the code. start_pc is inclusive. end_pc is exclusive.

handler_pc

This field gives the starting address of the exception handler. The value of the field is an offset from the start of the code.

catch_type

If catch_type is non-zero, then constant_pool[catch_type] will be the class of exceptions that this exception handler is designated to catch. This exception handler should only be called if the thrown exception is an instance of the given class.

If catch_type is zero, this exception handler should be called for all exceptions.

attributes_count

This value gives the number of additional attributes about code. The "Code" attribute can itself have attributes.

attributes

A "Code" attribute can have any number of optional attributes associated with it. Each attribute has a name, and other additional information. Currently, the only code attributes recognized are the "LineNumberTable" and "LocalVariableTable," both of which contain debugging information.

Any other attributes are skipped.

Line Number Table Attribute

The Line Number Table is used by debuggers and the exception handler to determine which part of the virtual machine code corresponds to a given location in the source. The LineNumberTable_attribute has the following format:

```
LineNumberTable_attribute {

        u2              attribute_name_index;

        u2              attribute_length;

        u2              line_number_table_length;
```

```
        {   u2                    start_pc;

                u2          line_number;

        }               line_number_table[line_number_table_length];

}
```

attribute_name_index

constant_pool[attribute_name_index] will be the string "LineNumberTable."

attribute_length

This field gives the total length of the LineNumberTable_attribute, excluding the initial four bytes.

line_number_table_length

This field gives the number of entries in the following line number table.

line_number_table

Each entry in the line number table indicates that the line number in the source file changes at a given point in the code.

start_pc

This field indicates the place in the code at which the code for a new line in the source begins. source_pc is an offset from the beginning of the code.

line_number

The line number that begins at the given location in the file.

Local Variable Table Attribute

The Local Variable Table is used by debuggers to determine the value of a given local variable during the dynamic execution of a method. The format of the LocalVariableTable_attribute is as follows:

```
LocalVariableTable_attribute {
```

```
        u2      attribute_name_index;

        u2      attribute_length;

        u2      local_variable_table_length;

        {  u2      start_pc;

           u2      length;

           u2      name_index;

           u2      signature_index;

           u2      slot;

        }       local_variable_table[local_variable_table_length];

}
```

attribute_name_index

constant_pool[attribute_name_index] will be the string "LocalVariableTable."

attribute_length

This field gives the total length of the LineNumberTable_attribute, excluding the initial four bytes.

local_variable_table_length

This field gives the number of entries in the following local variable table.

line_number_table

Each entry in the line number table indicates a code range during which a local variable has a value. It also indicates where on the stack the value of that variable can be found.

start_pc, length

The given local variable will have a value at the code between start_pc and start_pc + length. The two values are both offsets from the beginning of the code.

name_index, signature_index

constant_pool[name_index]and constant_pool[signature_index] are strings giving the name and signature of the local variable.

slot

The given variable will be the slotth local variable in the method's frame.

---

vmspec.: The Java Virtual Machine-Components of the Virtual Machine

---

The Java Virtual Machine

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 mach ine. 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 a re 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.

---

vmspec.: Class File Format- Constant Pool

Class File Format

Constant Pool

Each item in the constant pool begins with a 1-byte tag:. The table below lists the valid tags and their values.

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_InterfaceMethodref	11
CONSTANT_NameandType	12
CONSTANT_Asciz	1

Each tag byte is then followed by one or more bytes giving more information about the specific constant.

Strings

CONSTANT_Asciz and CONSTANT_Unicode are used to represent constant string values.

```
CONSTANT_Asciz_info {

        u1      tag;

        u2      length;

        u1      bytes[length];

}



CONSTANT_Unicode_info {

        u1      tag;

        u2      length;

        u2      bytes[length];

}
```

tag

The tag will have the value CONSTANT_Asciz or CONSTANT_Unicode.

length

The number of bytes in the string. This length does not include the implicit null termination.

bytes

The actual bytes in the string. The null termination is not included.

Classes and Interfaces

CONSTANT_Class is used to represent a class or an interface.

```
CONSTANT_Class_info {

        u1      tag;

        u2      name_index;

}
```

tag

The tag will have the value CONSTANT_Class

name_index

constant_pool[name_index] is a string giving the name of the class.

Because arrays are objects, the opcodes anewarray and multianewarray can reference array "classes" via CONSTANT_Class items in the constant pool. In this case, the name of the class is its signature. For example, the class name for

```
int[][]
```

is

```
[[I
```

The class name for

```
Thread[]
```

is

```
"[Ljava.lang.Thread;"
```

Fields and Methods

Fields, methods, and interface methods are represented by similar structures.

```
CONSTANT_Fieldref_info {

        u1      tag;

        u2      class_index;

        u2      name_and_type_index;

}
```

```
CONSTANT_Methodref_info {

        u1      tag;

        u2      class_index;

        u2      name_and_type_index;

}
```

```
CONSTANT_InterfaceMethodref_info {

        u1      tag;

        u2      class_index;

        u2      name_and_type_index;

}
```

tag

The tag will have the value CONSTANT_Fieldref, CONSTANT_Methodref, or CONSTANT_InterfaceMethodref.

class_index

constant_pool[class_index] will be an entry of type CONSTANT_Class giving the name of the class or interface containing the field or method.

For CONSTANT_Fieldref and CONSTANT_Methodref, the CONSTANT_Class item must be an actual class. For CONSTANT_InterfaceMethodref, the item must be an interface which purports to implement the given method.

name_and_type_index

constant_pool[name_and_type_index] will be an entry of type CONSTANT_NameAndType. This constant pool entry gives the name and signature of the field or method.

Abstract Fields and Methods

CONSTANT_NameAndType is used to represent a field or method, detached from any particular class or implementation.

```
CONSTANT_NameAndType_info {

        u1      tag;

        u2      name_index;

        u2      signature_index;

}
```

tag

The tag will have the value CONSTANT_NameAndType

name_index

constant_pool[name_index] is a string giving the name of the field or method.

signature_index

constant_pool[signature_index] is a string giving the signature of the field or method.

String Objects

CONSTANT_String is used to represent constant objects of the built-in type String.

```
CONSTANT_String_info {

        u1      tag;
```

```
        u2      string_index;

}
```

tag

The tag will have the value CONSTANT_String

name_index

constant_pool[string_index] is a string giving the value to which the String object is initialized.

The string at constant_pool[string_index] is "encoded" so that strings containing only ASCIZ characters, can be represented using only one byte per character, but characters of up to 16 bits can be represented. The format we use is a modified UTF *1 format.

All characters in the range 0x0001 to 0x007F are represented by a single byte:

```
+-+-+-+-+-+-+-+-+
|0|7bits of data|
+-+-+-+-+-+-+-+-+
```

The null character (0x0000) and characters in the range 0x0080 to 0x03FF are represented by a pair of two bytes:

```
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
|1|1|0| 5 bits |   |1|0|  6 bits   |
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
```

Characters in the range 0x0400 to 0xFFFF are represented by three bytes:

Numeric Constants

Four-Byte Constants

CONSTANT_Integer and CONSTANT_Float represent four-byte constants.

```
CONSTANT_Integer_info {

        u1      tag;

        u4      bytes;

}
```

```
CONSTANT_Float_info {

        u1      tag;

        u4      bytes;

}
```

tag

The tag will have the value CONSTANT_Integer or CONSTANT_Float

bytes

For integers, the four bytes are in the integer. For floats, the four bytes represent the standard IEEE representation of the floating point number.

Eight-Byte Constants

CONSTANT_Long and CONSTANT_Double represent eight-byte constants.

```
CONSTANT_Long_info {

        u1      tag;

        u4      high_bytes;

        u4      low_bytes;

}
```

```
CONSTANT_Double_info {

        u1      tag;

        u4      high_bytes;

        u4      low_bytes;

}
```

All eight-byte constants take up two spots in the constant pool. If this is the nth item in the constant pool, then the next item will be numbered n+2.

tag

The tag will have the value CONSANT_Long or CONSTANT_Double.

high_bytes, low_bytes

For CONSTANT_Long, the 64-bit value is (high_bytes <;<; 32) + low_bytes.

For CONSTANT_Double, the 64-bit value, high_bytes and low_bytes together represent the standard IEEE representation of the double-precision floating point number.

---

vmspec.: Class File Format-Signatures

Class File Format

Signatures

A signature is a string representing the type of a method or field.

The field signature represents the value of an argument to a function or the value of a variable. It is a series of bytes in the following grammar:

```
<;field signature>;     :=      <;field_type>;

<;field type>;  :=      <;base_type>;|<;object_type>;|<;array_type>;

<;base_type>;   :=      B|C|D|F|I|J|S|Z

<;object_type>; :=      L<;fullclassname>;;

<;array_type>;  :=      [<;optional-size>;<;field_type>;

<;optional_size>;       :=      [0-9]*
```

The meaning of the base types is as follows:

```
        B                       signed byte
        C                       character
        D                       double precision floating point number
        F                       single precision floating point number
        I                       integer
        J                       long integer
        L<;fullclassname>;;                     an object of the given class
        S                       nsigned short
```

```
Z                          boolean
[<;length>;<;field sig>;                          array
```

A return-type signature represents the return value from a method. It is a series of bytes in the following grammar:

```
<;return signature>;    :=      <;field type>; | V
```

The character V indicates that the method returns no value. Otherwise, the signature indicates the type of the return value.

An argument signature represents an argument passed to a method:

```
<;argument signature>;  :=      <;field type>;
```

A method signature represents the arguments that the method expects, and the value that it returns.

```
<;method_signature>;    :=      (<;arguments signature>;) <;return
signature>;
```

```
<;arguments signature>; :=      <;argument signature>;*
```

---

vmspec: - An Optimization

Appendix A - An Optimization

The following set of pseudo-instructions suffixed by _quick are variants of Java virtual machine instructions. They are used by the WebRunner/Java project to improve the execution of compiled code on our bytecode interpreter. They are not part of the virt ual machine specification or instruction set, and are invisible outside of an Java virtual machine implementation. However, inside a virtual machine implementation they have proven to be an effective optimization.

A compiler from Java to the Java virtual machine instruction set emits only non-_quick instructions. If the _quick pseudo-instructions are used, each instance of a non-_quick instruction with a _quick variant is overwritten on execution by its _quick vari ant. Subsequent execution of that instruction instance will be of the _quick variant.

In all cases, if an instruction has an alternative version with the suffix _quick, the instruction references the constant pool. If the _quick optimization is used, each non_quick instruction with a _quick variant performs the following:

- Resolves the specified item in the constant pool
- Signals an error if the item in the constant pool could not be resolved for some reason
- Turns itself into the _quick version of the instruction. The instructions putstatic, getstatic, putfield, and getfield each have two _quick versions.
- Performs its intended operation

This is identical to the action of the instruction without the _quick optimization, except for the additional step in which the instruction overwrites itself with its _quick variant.

The _quick variant of an instruction assumes that the item in the constant pool has already been resolved, and that this resolution did not generate any errors. It simply performs the intended operation on the resolved item.

---

vmspec.: - An Optimization- Pushing Constants onto the Stack (_quick v

---

- An Optimization

Pushing Constants onto the Stack (_quick variants)

---

ldc1_quick

Push item from constant pool onto stack
Syntax:

ldc1_quick = 199
indexbyte1

```
... =>; ..., item
```

indexbyte1 is used as an unsigned 8-bit index into the constant pool of the current class. The item at that index is pushed onto the stack.

ldc2_quick

Push item from constant pool onto stack
Syntax:

ldc2_quick = 200
indexbyte1
indexbyte2

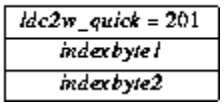
```
... =>; ..., item
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant at that index is resolved and the item at that index is pushed onto the stack.

ldc2w_quick

Push long integer or double float from constant pool onto stack

Syntax:




```
... =>;=>; ..., constant-word1, constant-word2
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant at that index is pushed onto the stack.

---

vmspec.: - An Optimization- Managing Arrays (_quick variants)

- An Optimization

Managing Arrays (_quick variants)

anewarray_quick

---

anewarray_quick

Allocate new array

Syntax:

anewarray_quick = 216
indexbyte1
indexbyte2

of objects

```
..., size =>; result
```

size should be an integer. It represents the number of elements in the new array.

indexbyte1 and indexbyte2 are are used to construct an index into the constant pool of the current class. The entry should be a class.

A new array of the indicated class type and capable of holding size elements is allocated. Allocation of an array large enough to contain nelem items of the given class type is attempted. All elements of the array are initialized to zero.

If size is less than zero, a NegativeArraySizeException is thrown. If there is not enough memory to allocate the array, an OutOfMemoryException is thrown.

---

vmspec.: - An Optimization- Manipulating Object Fields (_quick variant

---

- An Optimization

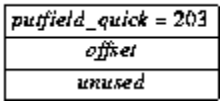
Manipulating Object Fields (_quick variants)

---

putfield_quick

Set field in object

Syntax:




```
..., handle, value =>; ...
```

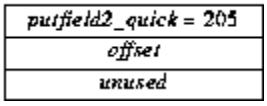
handle should be a handle to an object. value should be a value of a type appropriate for the specified field. offset is the offset for the field in that object. value is written at offset into the object referenced by handle. Both handle and value are popped from the stack.

If handle is null, a NullPointerException exception is generated.

putfield2_quick

Set long integer or double float field in object

Syntax:




```
..., handle, value-word1, value-word2=>; ...
```

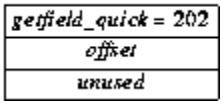
handle should be a handle to an object. value should be a value of a type appropriate for the specified field. offset is the offset for the field in that object. value is written at offset into the object referenced by handle. Both handle and value are popped from the stack.

If handle is null, a NullPointerException exception is generated.

getfield_quick

Fetch field from object

Syntax:




```
..., handle =>; ..., value
```

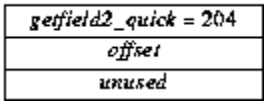
handle should be a handle to an object. The value at offset into the object referenced by handle replaces handle on the top of the stack.

If handle is null, a NullPointerException exception is generated.

getfield2_quick

Fetch field from object

Syntax:




```
..., handle =>; ..., value-word1, value-word2
```

handle should be a handle to an object. The value at offset into the object referenced by handle replaces handle on the top of the stack.

If handle is null, a NullPointerException exception is generated.

putstatic_quick

Set static field in class
Syntax:

putstatic_quick = 207
indexbyte1
indexbyte2

```
..., value =>; ...
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. value should be the type appropriate to that field. That field will be set to have the value value.

putstatic2_quick

Set static field in class
Syntax:

putstatic2_quick = 209
indexbyte1
indexbyte2

```
..., value-word1, value-word2 =>; ...
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. That field should either be a long integer or a double precision floating point number. value should be the type appropriate to that field. That field will be set to have the value value.

getstatic_quick

Get static field from class
Syntax:

getstatic_quick = 206
indexbyte1
indexbyte2

```
..., =>; ..., value
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The value of that field will replace handle on the stack.

getstatic2_quick

Get static field from class

Syntax:

getstatic2_quick = 208
indexbyte1
indexbyte2

```
..., =>; ..., value-word1, value-word2
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The field should be a long integer or a double precision floating point number. The value of that field will replace handle on the stack

---

vmspec.: - An Optimization- Method Invocation (_quick variants)

- An Optimization

Method Invocation (_quick variants)

---

invokevirtual_quick

Invoke class method

Syntax:

invokevirtual_quick = 210
offset
nargs

```
..., handle, [arg1, [arg2 ...]] =>; ...
```

The operand stack is assumed to contain a handle to an object and nargs arguments. The method block at offset in the object's method table is retrieved. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (nargs) expected on the operand stack.

If the method is marked synchronized the monitor associated with handle is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to handle on the stack, making handle and the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables use d by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a NullPointerException is thrown. If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

invokevirtualobject_quick

Invoke class method
Syntax:

invokevirtualobject_quick = 214
offset
nargs

```
..., handle, [arg1, [arg2 ...]] =>; ...
```

The operand stack is assumed to contain a handle to an object or to an array and nargs arguments. The method block at offset in the object's method table is retrieved. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (nargs) expected on the operand stack.

If the method is marked synchronized the monitor associated with handle is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to handle on the stack, making handle and the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables use d by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a NullPointerException is thrown. If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

invokenonvirtual_quick

Invoke superclass method
Syntax:

invokenonvirtual_quick = 211
indexbyte1
indexbyte2

```
..., handle, [arg1, [arg2 ...]] =>; ...
```

The operand stack is assumed to contain a handle to an object and some number of arguments. indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a method slot index and a pointer to a class. The method block at the method slot index in the indicated class is retrieved. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (nargs) expected on the operand stack.

If the method is marked synchronized the monitor associated with handle is entered. The exact behavior

of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to handle on the stack, making handle and the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables use d by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a NullPointerException is thrown. If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

invokestatic_quick

Invoke a static method

Syntax:

invokenstatic_quick = 212
indexbyte1
indexbyte2

```
..., [arg1, [arg2 ...]] =>; ...
```

The operand stack is assumed to contain some number of arguments. indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a method slot index and a pointer to a class. The method block at the method slot index in the indicated class is retrieved. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (nargs) expected on the operand stack.

If the method is marked synchronized the monitor associated with the method's class is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to the first argument on the stack, making the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables us ed by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a NullPointerException is thrown. If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

invokeinterface_quick

Invoke interface method

Syntax:

invokeinterface_quick = 213
idbyte1
idbyte2
nargs
guess

```
..., handle, [arg1, [arg2 ...]] =>; ...
```

The operand stack is assumed to contain a handle to an object and nargs-1 arguments. idbyte1 and idbyte2 are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object handle.

The method signature is searched for in the object's method table. As a short-cut, the method signature at slot guess is searched first. If that fails, a complete search of the method table is performed. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (native, synchronized, etc.) but unlike invokemethod and invokesuper, the number of available arguments (nargs) is taken from the bytecode.

If the method is marked synchronized the monitor associated with handle is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to handle on the stack, making handle and the supplied arguments (arg1, arg2, ...) the first nargs local variables of the new frame. The total number of local variables use d by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a NullPointerException is thrown. If during the method invocation a stack overflow is detected, a StackOverflowException is thrown.

guess is the last guess. Each time through, guess is set to the method offset that was used.

---

vmspec.: - An Optimization- Miscellaneous Object Operations (_quick va

---

- An Optimization

Miscellaneous Object Operations (_quick variants)

---

new_quick

Create new object

Syntax:

new_quick = 215
indexbyte1
indexbyte2

```
... =>; ..., handle
```

indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item at that index should be a class. A new instance of that class is then created and a handle for it pushed on the stack.

checkcast_quick

Make sure object is of given type

Syntax:

checkcast_quick = 217
indexbyte1
indexbyte2

```
..., handle =>; ..., handle
```

handle should be a handle to an object. indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The object at that index of the constant pool should have already been resolved.

checkcast then determines whether handle can be cast to an object of class class. A null handle can be cast to any class, and otherwise the superclasses of handle are searched for class. If class is determined to be a superclass of handle, or if handle is null, object can be cast to class and execution proceeds at the next instruction, and the handle for handle remains on the stack.

If handle cannot be cast to class, a ClassCastException is thrown.

instanceof_quick

Syntax:

instanceof_quick = 218
indexbyte1
indexbyte2

Determine if object is of given type

```
..., handle =>; ..., result
```

handle should be a handle to an object. indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current class. The item of class class at that index of the constant pool is assumed to have already been resolved.

instanceof determines whether handle can be cast to an object of the class class. A null handle can be cast to any class, and otherwise the superclasses of handle are searched for class. If class is determined to be a superclass of handle, or if handle is null, handle is overwritten by 1. Otherwise, handle is overwritten by 0.

---

vmspec.: - An Optimization- Constant Pool Resolution

- An Optimization

Constant Pool Resolution

When the class is read in, an array constant_pool[] of size nconstants is created and assigned to a field in the class. constant_pool[0] is set to point to a malloc-ed array which indicates which fields in the constant_pool have already been resolved. con stant_pool[1] through constant_pool[nconstants - 1] are set to point at the "type" field that corresponds to this constant item.

When an instruction is executed that references the constant pool, an index is generated, and constant_pool[0] is checked to see if the index has already been resolved. If so, the value of constant_pool[index] is returned. If not, the value of constant_po ol[index] is resolved to be the actual pointer or data, and overwrites whatever value was already in constant_pool[index].

---

vmspec.: The Java Virtual Machine-The Java Instruction Set

The Java Virtual Machine

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 t he 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 lo okupswitch 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.

---

vmspec.: The Java Virtual Machine-Primitive Data Types

The Java Virtual Machine

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 differe nt 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.

---

vmspec.: The Java Virtual Machine-Registers

The Java Virtual Machine

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).

---

vmspec.: The Java Virtual Machine-The Java Stack

The Java Virtual Machine

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 progra mming 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 ca tch 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.

---

vmspec.: The Java Virtual Machine-Operand Stack

The Java Virtual Machine

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 wor ds 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.

URL Not Available

http://www.cern.ch/

URL Not Available

http://www.cern.ch/CERN/Divisions/ECP/PT/Welcome.html

URL Not Available

http://www.cern.ch/WebMaker/