2 The Virtual Machine Instruction Set

Pushing Constants onto the Stack

bipush

Push one-byte signed integer
Syntax:

```
bipush = 16
byte1
```

Stack: ... => ..., 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:

```
sipush = 17
byte1
byte2
```

Stack: ... => ..., `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:

```
ldc1 = 18
indexbyte1
```

Stack: ... => ..., `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.
The Virtual Machine Instruction Set

ldc2
Push item from constant pool
Syntax:

\[
\begin{array}{c}
\text{ldc2} = 19 \\
\text{indexbyte1} \\
\text{indexbyte2}
\end{array}
\]

Stack: \( \ldots \Rightarrow \ldots, \text{item} \)

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:

\[
\begin{array}{c}
\text{ldc2w} = 20 \\
\text{indexbyte1} \\
\text{indexbyte2}
\end{array}
\]

Stack: \( \ldots \Rightarrow \ldots, \text{constant-word1, constant-word2} \)

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:

\[
\begin{array}{c}
\text{aconst_null} = 1
\end{array}
\]

Stack: \( \ldots \Rightarrow \ldots, \text{null} \)

Push the null object onto the stack.

iconst_m1
Push integer constant -1
Syntax:

\[
\begin{array}{c}
\text{iconst_m1} = 2
\end{array}
\]

Stack: \( \ldots \Rightarrow \ldots, -1 \)

Push the integer -1 onto the stack.

iconst_<n>
Push integer constant <n>
Syntax:

\[
\begin{array}{c}
\text{iconst_<n>}
\end{array}
\]

Stack: \( \ldots \Rightarrow \ldots, \text{<n>} \)

Forms: \text{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.
The Virtual Machine Instruction Set

lconst_<l>
Push long integer constant
Syntax:

Stack: ... => ..., <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:

Stack: ... => ..., <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:

Stack: ... => ..., <d>-word1, <d>-word2
Forms: dconst_0 = 14, dconst_1 = 15
Push the double precision floating point number <d> onto the stack.

Loading Local Variables Onto the Stack

iload
Load integer from local variable
Syntax:

Stack: ... => ..., value
Local variable vindex in the current Java frame should contain an integer. The value of that variable is pushed onto the operand stack.
The Virtual Machine Instruction Set

**iload_<n>**
Load integer from local variable

**Syntax:**
```
  iload_<n>
```

**Stack:** ... => ..., 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:**
```
  lload = 22
  vindex
```

**Stack:** ... => ..., value-word1, value-word2

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

**lload_<n>**
Load long integer from local variable

**Syntax:**
```
  lload_<n>
```

**Stack:** ... => ..., 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:**
```
  fload = 23
  vindex
```

**Stack:** ... => ..., 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.
The Virtual Machine Instruction Set

**fload\_\textless n\textgreater**
Load single float from local variable
Syntax:

```
fload\_\textless n\textgreater
```

Stack: \(... \Rightarrow ...\), \textit{value}
Forms: \textit{fload\_0} = 34, \textit{fload\_1} = 35, \textit{fload\_2} = 36, \textit{fload\_3} = 37
Local variable \textit{\textless n\textgreater} in the current Java frame should contain a single precision floating point number. The \textit{value} of that variable is pushed onto the operand stack.
This opcode is the same as \textit{fload} with a \textit{vindex} of \textit{\textless n\textgreater}, except that the operand \textit{\textless n\textgreater} is implicit.

**dload**
Load double float from local variable
Syntax:

```
dload = 24
vindex
```

Stack: \(... \Rightarrow ...\), \textit{value-word1}, \textit{value-word2}
Local variables \textit{vindex} and \textit{vindex+1} in the current Java frame should together contain a double precision float point number. The \textit{value} contained in those variables is pushed onto the operand stack.

**dload\_\textless n\textgreater**
Load double float from local variable
Syntax:

```
dload\_\textless n\textgreater
```

Stack: \(... \Rightarrow ...\), \textit{value-word1}, \textit{value-word2}
Forms: \textit{dload\_0} = 38, \textit{dload\_1} = 39, \textit{dload\_2} = 40, \textit{dload\_3} = 41
Local variables \textit{\textless n\textgreater} and \textit{\textless n\textgreater+1} in the current Java frame should together contain a double precision floating point number. The \textit{value} contained in those variables is pushed onto the operand stack.
This opcode is the same as \textit{dload} with a \textit{vindex} of \textit{\textless n\textgreater}, except that the operand \textit{\textless n\textgreater} is implicit.

**aload**
Load local object variable
Syntax:

```
aload = 25
vindex
```

Stack: \(... \Rightarrow ...\), \textit{value}
Local variable \textit{vindex} in the current Java frame should contain a handle to an object or to an array. The \textit{value} of that variable is pushed onto the operand stack.
The Virtual Machine Instruction Set

aload_<n>
Load object reference from local variable
Syntax:

\[
\text{aload}_<n>
\]
Stack: \(... \Rightarrow ..., \text{value}\)
Forms: \text{aload}_0 = 42, \text{aload}_1 = 43, \text{aload}_2 = 44, \text{aload}_3 = 45\nLocal variable \(n\) in the current Java frame should contain a handle to an object or to an array. The \textit{value} of that variable is pushed onto the operand stack.
This opcode is the same as \textit{aload} with a \textit{vindex} of \(<n>\), except that the operand \(<n>\) is implicit.

Storing Stack Values into Local Variables

istore
Store integer into local variable
Syntax:

\[
\text{istore} = 54 \quad \text{vindex}
\]
Stack: \(..., \text{value} \Rightarrow ...\)
\textit{value} should be an integer. Local variable \textit{vindex} in the current Java frame is set to \textit{value}.

istore_<n>
Store integer into local variable
Syntax:

\[
\text{istore}_<n>
\]
Stack: \(..., \text{value} \Rightarrow ...\)
Forms: \text{istore}_0 = 59, \text{istore}_1 = 60, \text{istore}_2 = 61, \text{istore}_3 = 62\n\textit{value} should be an integer. Local variable \(<n>\) in the current Java frame is set to \textit{value}.
This instruction is the same as \textit{istore} with a \textit{vindex} of \(<n>\), except that the operand \(<n>\) is implicit.

Lstore
Store long integer into local variable
Syntax:

\[
\text{Lstore} = 55 \quad \text{vindex}
\]
Stack: \(..., \text{value-word1}, \text{value-word2} \Rightarrow ...\)
\textit{value} should be a long integer. Local variables \textit{vindex} and \textit{vindex+1} in the current Java frame are set to \textit{value}.
The Virtual Machine Instruction Set

\textbf{lstore}_{<n>}

Store long integer into local variable

Syntax:

\begin{verbatim}
lstore_{<n>}
\end{verbatim}

Stack: \ldots, \textit{value-word1, value-word2} \Rightarrow \ldots

Forms: lstore\_0 = 63, lstore\_1 = 64, lstore\_2 = 65, lstore\_3 = 66

\textit{value} should be a long integer. Local variables \(<n>\) and \(<n>+1\) in the current Java frame are set to \textit{value}.

This instruction is the same as \textit{lstore} with a \textit{vindex} of \(<n>\), except that the operand \(<n>\) is implicit.

\textbf{fstore}

Store single float into local variable

Syntax:

\begin{verbatim}
fstore = 56
\end{verbatim}

\begin{verbatim}
  vindex
\end{verbatim}

Stack: \ldots, \textit{value} \Rightarrow \ldots

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

\textbf{fstore}_{<n>}

Store single float into local variable

Syntax:

\begin{verbatim}
fstore_{<n>}
\end{verbatim}

Stack: \ldots, \textit{value} \Rightarrow \ldots

Possible Instructions:

\begin{verbatim}
fstore\_0 = 67, fstore\_1 = 68, fstore\_2 = 69, fstore\_3 = 70
\end{verbatim}

\textit{value} should be a single precision floating point number. Local variable \(<n>\) in the current Java frame is set to \textit{value}.

This instruction is the same as \textit{fstore} with a \textit{vindex} of \(<n>\), except that the operand \(<n>\) is implicit.

\textbf{dstore}

Store double float into local variable

Syntax:

\begin{verbatim}
dstore = 57
\end{verbatim}

\begin{verbatim}
  vindex
\end{verbatim}

Stack: \ldots, \textit{value-word1, value-word2} \Rightarrow \ldots

\textit{value} should be a double precision floating point number. Local variables \textit{vindex} and \textit{vindex}+1 in the current Java frame are set to \textit{value}.
The Virtual Machine Instruction Set

dstore_<n>
Store double float into local variable
Syntax:

```
dstore_<n>
```
Stack: ..., value-word1, value-word2 => ...
Forms: dstore_0 = 71, dstore_1 = 72, dstore_2 = 73, dstore_3 = 74
value should be a 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:

```
astore = 58
vindex
```
Stack: ..., 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>
Store object reference into local variable
Syntax:

```
astore_<n>
```
Stack: ..., 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:

```
iinc = 132
vindex
const
```
Stack: 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.
Managing Arrays

**newarray**

Allocate new array

Syntax:

<table>
<thead>
<tr>
<th>newarray = atype</th>
</tr>
</thead>
</table>

Stack: ..., 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.
The Virtual Machine Instruction Set

**anewarray**

Allocate new array of objects

Syntax:

```
anewarray = 189
indexbyte1
indexbyte2
```

Stack: `... , size => result`

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

`indexbyte1` and `indexbyte2` 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:

```
multianewarray = 197
indexbyte1
indexbyte2
dimensions
```

Stack: `... , 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 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 `≥ 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:
The Virtual Machine Instruction Set

```java
new int[6][3][]
```
the following code is used:
```java
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
Get length of array
Syntax:
```java
arraylength = 190
```
Stack: ..., 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:
```java
iaload = 46
```
Stack: ..., 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.

### iload
Load long integer from array
Syntax:
```java
iload = 47
```
Stack: ..., 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.
The Virtual Machine Instruction Set

faload
Load single float from array
Syntax:

\[
\text{faload} = 48
\]

Stack: ..., 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:

\[
\text{daload} = 49
\]

Stack: ..., 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:

\[
\text{aaload} = 50
\]

Stack: ..., 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:

\[
\text{baload} = 51
\]

Stack: ..., 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.
The Virtual Machine Instruction Set

**caload**

Load character from array

Syntax:

```
caload = 52
```

Stack: ..., 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:

```
saload = 53
```

Stack: ..., 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:

```
iastore = 79
```

Stack: ..., 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:

```
lastore = 80
```

Stack: ..., 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.
The Virtual Machine Instruction Set

fastore

Store into single float array

Syntax:

\[ \text{fastore} = 81 \]

Stack: \(..., \text{array}, \text{index}, \text{value} \Rightarrow ...\)

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:

\[ \text{dastore} = 82 \]

Stack: \(..., \text{array}, \text{index}, \text{value-word1}, \text{value-word2} \Rightarrow ...\)

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:

\[ \text{aastore} = 83 \]

Stack: \(..., \text{array}, \text{index}, \text{value} \Rightarrow ...\)

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 an 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:

\[ \text{bastore} = 84 \]

Stack: \(..., \text{array}, \text{index}, \text{value} \Rightarrow ...\)

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.
The Virtual Machine Instruction Set

**castore**

Store into character array

Syntax:

```
castore = 85
```

Stack: `...`, `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**

Store into short array

Syntax:

```
sastore = 86
```

Stack: `...`, `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 a 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.

**Stack Instructions**

**nop**

Do nothing.

Syntax:

```
nop = 0
```

Stack: no change

Do nothing.

**pop**

Pop top stack word

Syntax:

```
pop = 87
```

Stack: `...`, `any` => `...`

Pop the top word from the stack.
pop2
Pop top two stack words
Syntax: \[ \text{pop2} = 88 \]
Stack: \(\ldots, \text{any2}, \text{any1} \Rightarrow \ldots\)
Pop the top two words from the stack.

dup
Duplicate top stack word
Syntax: \[ \text{dup} = 89 \]
Stack: \(\ldots, \text{any} \Rightarrow \ldots, \text{any}, \text{any}\)
Duplicate the top word on the stack.

dup2
Duplicate top two stack words
Syntax: \[ \text{dup2} = 92 \]
Stack: \(\ldots, \text{any2}, \text{any1} \Rightarrow \ldots, \text{any2}, \text{any1}, \text{any2}, \text{any1}\)
Duplicate the top two words on the stack.

dup_x1
Duplicate top stack word and put two down
Syntax: \[ \text{dup_x1} = 90 \]
Stack: \(\ldots, \text{any2}, \text{any1} \Rightarrow \ldots, \text{any1}, \text{any2}, \text{any1}\)
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: \[ \text{dup2_x1} = 93 \]
Stack: \(\ldots, \text{any3}, \text{any2}, \text{any1} \Rightarrow \ldots, \text{any2}, \text{any1}, \text{any3}, \text{any2}, \text{any1}\)
Duplicate the top two words on the stack and insert the copies two words down in the stack.
The Virtual Machine Instruction Set

**dup\_x2**
Duplicate top stack word and put three down.

Syntax:  
```plaintext
dup\_x2 = 91
```

Stack:  `...\text{,...,any3,any2,any1}\Rightarrow...\text{,...,any1,any3,any2,any1}`
Duplicate the top word on the stack and insert the copy three words down in the stack.

**dup\_2\_x2**
Duplicate top two stack words and put three down

Syntax:  
```plaintext
dup2\_x2 = 94
```

Stack:  `...\text{,...,any4,any3,any2,any1}\Rightarrow...\text{,...,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:  
```plaintext
swap = 95
```

Stack:  `...\text{,...,any2,any1}\Rightarrow...\text{,...,any2,any1}`
Swap the top two elements on the stack.

**Arithmetic Instructions**

**iadd**
Integer add

Syntax:  
```plaintext
iadd = 96
```

Stack:  `...\text{,...,value1,value2}\Rightarrow...\text{,...,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:  
```plaintext
ladd = 97
```

Stack:  `...\text{,...,value1-word1,value1-word2,value2-word1,value2-word2}\Rightarrow...\text{,...,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.
The Virtual Machine Instruction Set

**fadd**
Single float add
Syntax:
```
fadd = 98
```
Stack: ..., 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
```
Stack: ..., 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:
```
isub = 100
```
Stack: ..., 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:
```
lsub = 101
```
Stack: ..., 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:
```
fsub = 102
```
Stack: ..., 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.
The Virtual Machine Instruction Set

**dsub**

Double float subtract

Syntax:

```
\textit{dsub} = 103
```

Stack: \(\ldots, \text{value1-word1}, \text{value1-word2}, \text{value2-word1}, \text{value2-word2} \Rightarrow \ldots, \text{result-word1}, \text{result-word2}\)

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

**imul**

Integer multiply

Syntax:

```
imul = 104
```

Stack: \(\ldots, \text{value1}, \text{value2} \Rightarrow \ldots, \text{result}\)

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

**lmul**

Long integer multiply

Syntax:

```
lmul = 105
```

Stack: \(\ldots, \text{value1-word1}, \text{value1-word2}, \text{value2-word1}, \text{value2-word2} \Rightarrow \ldots, \text{result-word1}, \text{result-word2}\)

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

**fmul**

Single float multiply

Syntax:

```
fmul = 106
```

Stack: \(\ldots, \text{value1}, \text{value2} \Rightarrow \ldots, \text{result}\)

\textit{value1} and \textit{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:

```
dmul = 107
```

Stack: \(\ldots, \text{value1-word1}, \text{value1-word2}, \text{value2-word1}, \text{value2-word2} \Rightarrow \ldots, \text{result-word1}, \text{result-word2}\)

\textit{value1} and \textit{value2} should be double precision floating point numbers. Both values are replaced on the stack by their double precision floating point product.
The Virtual Machine Instruction Set

**idiv**

Integer divide

Syntax:  

\[
\text{idiv} = 108
\]

Stack:  ... , 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:  

\[
\text{ldiv} = 109
\]

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

\[
\text{fdiv} = 110
\]

Stack:  ... , value, 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:  

\[
\text{ddiv} = 111
\]

Stack:  ... , value-word1, value-word2, value-word1, value-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.
The Virtual Machine Instruction Set

**imod**

Integer mod

Syntax:

```
imod = 112
```

Stack: ..., 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:

```
lmod = 113
```

Stack: ..., 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:

```
fmod = 114
```

Stack: ..., 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 floating 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:

```
dmod = 115
```

Stack: ..., 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 floating 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.
The Virtual Machine Instruction Set

ineg
Integer negate
Syntax:

\[
\text{ineg} = 116
\]
Stack: ..., value => ..., result
value should be an integer. It is replaced on the stack by its arithmetic negation.

lneg
Long integer negate
Syntax:

\[
\text{lneg} = 117
\]
Stack: ..., 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:

\[
\text{fneg} = 118
\]
Stack: ..., 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:

\[
\text{dneg} = 119
\]
Stack: ..., 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.

Logical Instructions

ishl
Integer shift left
Syntax:

\[
\text{ishl} = 120
\]
Stack: ..., 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.
The Virtual Machine Instruction Set

ishr
Integer arithmetic shift right
Syntax:

\[ ishr = 122 \]

Stack: ..., 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:

\[ iushr = 124 \]

Stack: ..., 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:

\[ lshl = 121 \]

Stack: ..., 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
Long integer arithmetic shift right
Syntax:

\[ lshr = 123 \]

Stack: ..., 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:

\[ lushr = 125 \]

Stack: ..., 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.
The Virtual Machine Instruction Set

**iand**

Integer boolean and
Syntax:

\[
\text{iand} = 126
\]

Stack: ..., 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:

\[
\text{land} = 127
\]

Stack: ..., 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:

\[
\text{ior} = 128
\]

Stack: ..., 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:

\[
\text{lor} = 129
\]

Stack: ..., 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:

\[
\text{ixor} = 130
\]

Stack: ..., value1, value2 => ..., result

`value1` and `value2` should both be integers. They are replaced on the stack by their bitwise exclusive disjunction (XOR).
The Virtual Machine Instruction Set

lxor
Long integer boolean xor
Syntax:  

Stack: ..., 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).

Conversion Operations

i2l
Integer to long integer conversion
Syntax:  

Stack: ..., 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:  

Stack: ..., 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:  

Stack: ..., 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:  

Stack: ..., value-word1, value-word2 => ..., result
value should be a long integer. It is converted to a integer. The result replaces value on the stack.
The Virtual Machine Instruction Set

**l2f**

Long integer to single float

Syntax:

```
l2f = 137
```

Stack: ..., 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:

```
l2d = 138
```

Stack: ..., 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:

```
f2i = 139
```

Stack: ..., 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:

```
f2l = 140
```

Stack: ..., 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
```

Stack: ..., 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.
The Virtual Machine Instruction Set

**d2i**

Double float to integer

Syntax:

```
d2i = 142
```

Stack: ..., 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
```

Stack: ..., 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
```

Stack: ..., 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:

```
int2byte = 145
```

Stack: ..., 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**

Integer to char

Syntax:

```
int2char = 146
```

Stack: ..., <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.
The Virtual Machine Instruction Set

int2short
Integer to char
Syntax: \[ \text{int2short} = 147 \]
Stack: ... , <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.

Control Transfer Instructions

ifeq
Branch if equal to 0
Syntax: \[
\begin{array}{l}
\text{ifeq} = 153 \\
\text{branchbyte1} \\
\text{branchbyte2}
\end{array}
\]
Stack: ... , 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 0
Syntax: \[
\begin{array}{l}
\text{iflt} = 155 \\
\text{branchbyte1} \\
\text{branchbyte2}
\end{array}
\]
Stack: ... , 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 to 0
Syntax: \[
\begin{array}{l}
\text{ifle} = 158 \\
\text{branchbyte1} \\
\text{branchbyte2}
\end{array}
\]
Stack: ... , 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.
The Virtual Machine Instruction Set

ifne
Branch if not equal to 0
Syntax:

```
ifne = 154
branchbyte1
branchbyte2
```

Stack: ..., 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 0
Syntax:

```
ifgt = 157
branchbyte1
branchbyte2
```

Stack: ..., 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 to 0
Syntax:

```
ifge = 156
branchbyte1
branchbyte2
```

Stack: ..., 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:

```
if_icmpeq = 159
branchbyte1
branchbyte2
```

Stack: ..., 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`. 
The Virtual Machine Instruction Set

if icmpne
Branch if integers not equal
Syntax:

\[
\text{if \_icmpne} = 160 \\
\text{branchbyte1} \\
\text{branchbyte2}
\]

Stack: \( \ldots, \text{value1}, \text{value2} \Rightarrow \ldots \)

\( \text{value1} \) and \( \text{value2} \) should be integers. They are both popped from the stack. If \( \text{value1} \) is not equal to \( \text{value2} \), \( \text{branchbyte1} \) and \( \text{branchbyte2} \) 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:

\[
\text{if \_icmplt} = 161 \\
\text{branchbyte1} \\
\text{branchbyte2}
\]

Stack: \( \ldots, \text{value1}, \text{value2} \Rightarrow \ldots \)

\( \text{value1} \) and \( \text{value2} \) should be integers. They are both popped from the stack. If \( \text{value1} \) is less than \( \text{value2} \), \( \text{branchbyte1} \) and \( \text{branchbyte2} \) 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:

\[
\text{if \_icmpgt} = 163 \\
\text{branchbyte1} \\
\text{branchbyte2}
\]

Stack: \( \ldots, \text{value1}, \text{value2} \Rightarrow \ldots \)

\( \text{value1} \) and \( \text{value2} \) should be integers. They are both popped from the stack. If \( \text{value1} \) is greater than \( \text{value2} \) (C’s >), \( \text{branchbyte1} \) and \( \text{branchbyte2} \) 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:

\[
\text{if \_icmple} = 164 \\
\text{branchbyte1} \\
\text{branchbyte2}
\]

Stack: \( \ldots, \text{value1}, \text{value2} \Rightarrow \ldots \)

\( \text{value1} \) and \( \text{value2} \) should be integers. They are both popped from the stack. If \( \text{value1} \) is less than or equal to \( \text{value2} \), \( \text{branchbyte1} \) and \( \text{branchbyte2} \) 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.
The Virtual Machine Instruction Set

if_icmpge
Branch if integer greater than or equal to

Syntax:
\[
\begin{align*}
\text{if	extunderscore icmpge} &= 162 \\
\text{branchbyte1} & \\
\text{branchbyte2}
\end{align*}
\]

Stack: ..., 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:
\[
\text{lcmp} = 148
\]

Stack: ..., 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:
\[
\text{fcmpl} = 149
\]

Stack: ..., 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:
\[
\text{fcmpg} = 150
\]

Stack: ..., 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.
The Virtual Machine Instruction Set

**dcmpl**
Double float compare (-1 on incomparable)
Syntax:

```
dcmpl = 151
```

Stack: ..., 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:

```
dcmpg = 152
```

Stack: ..., 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:

```
if_acmpeq = 165
```

Stack: ..., 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:

```
if_acmpne = 166
```

Stack: ..., 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`. 
The Virtual Machine Instruction Set

**goto**
Branch always
Syntax:

```
goto = 167
branchbyte1
branchbyte2
```

Stack: 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:

```
jsr = 168
branchbyte1
branchbyte2
```

Stack: ... => ..., 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:

```
ret = 169
vindex
```

Stack: 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.
Function Return

ireturn
Return integer from function
Syntax:

```
ireturn = 172
```

Stack: ..., 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:

```
lreturn = 173
```

Stack: ..., 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:

```
freturn = 174
```

Stack: ..., 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:

```
dreturn = 175
```

Stack: ..., 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:
```
areturn = 176
```
Stack: ..., 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:
```
return = 177
```
Stack: ... => [empty]
All values on the operand stack are discarded. The interpreter then returns control to its caller.

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...
```
Stack: ..., 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:

```plaintext
lookupswitch = 171
  ...0-3 byte pad...
  default-offset1
default-offset2
default-offset3
default-offset4
  npairs1
  npairs2
  npairs3
  npairs4
  ...match-offset pairs..
```

Stack: ..., 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.

**Manipulating Object Fields**

**putfield**

Set field in object

Syntax:

```plaintext
putfield = 181
  indexbyte1
  indexbyte2
```

Stack: ..., handle, value => ...

OR

Stack: ..., 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.
The Virtual Machine Instruction Set

**getfield**

Fetch field from object

Syntax:

```
getfield = 180
indexbyte1
indexbyte2
```

Stack: ..., handle => ..., value

OR

Stack: ..., 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:

```
putstatic = 179
indexbyte1
indexbyte2
```

Stack: ..., value => ...

OR

Stack: ..., 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.
The Virtual Machine Instruction Set

getstatic

Get static field from class

Syntax:

<table>
<thead>
<tr>
<th>getstatic = 178</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

Stack: ..., => ..., value

OR

Stack: ..., => ..., 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.
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 multiply-inherited virtual methods in C++.

invokevirtual

Invoke class method

Syntax:

<table>
<thead>
<tr>
<th>invokevirtual</th>
<th>182</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
<td></td>
</tr>
<tr>
<td>indexbyte2</td>
<td></td>
</tr>
</tbody>
</table>

Stack: ..., 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 contains 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 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 used 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.
The Virtual Machine Instruction Set

**invokenonvirtual**

Invoke non-virtual method

Syntax:

```
invokenonvirtual = 183
indexbyte1
indexbyte2
```

Stack: ..., 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 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 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 used 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
```

Stack: ..., 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 used 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.
The Virtual Machine Instruction Set

**invokeinterface**

Invoke interface method

Syntax:

```
invokeinterface = 185
indexbyte1
indexbyte2
nargs
reserved
```

Stack: ..., 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 method 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 used 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.
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

Throw exception

Syntax:

```plaintext
athrow = 191
```

Stack: ..., 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 object 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.

Miscellaneous Object Operations

new

Create new object

Syntax:

```plaintext
new = 187
indexbyte1
indexbyte2
```

Stack: ... => ..., 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 from name

Syntax:

```plaintext
newfromname = 186
```

Stack: ..., 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:

```
checkcast = 192
indexbyte1
indexbyte2
```

Stack: ..., 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

Determine if object is of given type

Syntax:

```
instanceof = 193
indexbyte1
indexbyte2
```

Stack: ..., 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

Verify stack empty

Syntax:

```
verifystack = 196
```

Stack: ... => [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.
Monitors

monitorenter
Enter monitored region of code
Syntax: 
\[
\text{monitorenter} = 194
\]
Stack: \(..., \text{handle} \Rightarrow ...
\text{handle} \text{ should be a handle to an object.}

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

monitorexit
Exit monitored region of code
Syntax: 
\[
\text{monitorexit} = 195
\]
Stack: \(..., \text{handle} \Rightarrow ...
\text{handle} \text{ should be a handle to an object.}

The lock on \text{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 \text{handle} to be free are allowed to proceed.

Debugging

breakpoint
Call breakpoint handler
Syntax: 
\[
\text{breakpoint} = 198
\]

The \text{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 \text{breakpoint} instruction is removed.