### aaload

**Operation**  
Load reference from array

**Format**  
```
| aaload |
```

**Forms**  
```
aaload = 50 (0x32)
```

**Operand**  
```
..., arrayref, index ⇒
```

**Stack**  
```
..., value
```

**Description**  
The `arrayref` must be of type `reference` and must refer to an array whose components are of type `reference`. The `index` must be of type `int`. Both `arrayref` and `index` are popped from the operand stack. The reference `value` in the component of the array at `index` is retrieved and pushed onto the operand stack.

**Runtime Exceptions**  
If `arrayref` is null, `aaload` throws a `NullPointerException`. Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `aaload` instruction throws an `ArrayIndexOutOfBoundsException`.
**aastore**

**Operation**  Store into reference array

**Format**  

<table>
<thead>
<tr>
<th>aastore</th>
</tr>
</thead>
</table>

**Forms**  

`aastore = 83 (0x53)`

**Operand**  

`..., arrayref, index, value ⇒`

**Stack**  

`...`

**Description**  The `arrayref` must be of type `reference` and must refer to an array whose components are of type `reference`. The `index` must be of type `int` and `value` must be of type `reference`. The `arrayref`, `index`, and `value` are popped from the operand stack. The `reference` `value` is stored as the component of the array at `index`.

At run-time, the type of `value` must be compatible with the type of the components of the array referenced by `arrayref`. Specifically, assignment of a value of reference type `S` (source) to an array component of reference type `T` (target) is allowed only if:

- If `S` is a class type, then:
  - If `T` is a class type, then `S` must be the same class (§2.8.1) as `T`, or `S` must be a subclass of `T`;
  - If `T` is an interface type, `S` must implement (§2.13) interface `T`.

- If `S` is an interface type, then:
  - If `T` is a class type, then `T` must be `Object` (§2.4.7).
  - If `T` is an interface type, then `T` must be the same interface as `S` or a superinterface of `S` (§2.13.2).
**aastore (cont.)**

- If \( S \) is an array type, namely, the type \( SC[] \), that is, an array of components of type \( SC \), then:
  - If \( T \) is a class type, then \( T \) must be \( \text{Object} \) (§2.4.7).
  - If \( T \) is an array type \( TC[] \), that is, an array of components of type \( TC \), then one of the following must be true:
    - \( TC \) and \( SC \) are the same primitive type (§2.4.1).
    - \( TC \) and \( SC \) are reference types (§2.4.6), and type \( SC \) is assignable to \( TC \) by these runtime rules.
  - If \( T \) is an interface type, \( T \) must be one of the interfaces implemented by arrays (§2.15).

**Runtime Exceptions**

If \( \text{arrayref} \) is \( \text{null} \), \( \text{aastore} \) throws a \( \text{NullPointerException} \).

Otherwise, if \( \text{index} \) is not within the bounds of the array referenced by \( \text{arrayref} \), the \( \text{aastore} \) instruction throws an \( \text{ArrayIndexOutOfBoundsException} \).

Otherwise, if \( \text{arrayref} \) is not \( \text{null} \) and the actual type of \( \text{value} \) is not assignment compatible (§2.6.7) with the actual type of the components of the array, \( \text{aastore} \) throws an \( \text{ArrayStoreException} \).
**acast_null**

**Operation**  Push null

**Format**  

**Forms**  

\[ \text{acast\_null} = 1 \ (0x1) \]

**Operand**  

**Stack**  

... \Rightarrow ...

... null

**Description**  Push the null object reference onto the operand stack.

**Notes**  The Java virtual machine does not mandate a concrete value for null.
**aloa**

**Operation**  Load reference from local variable

**Format**  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aloa</strong></td>
<td><strong>index</strong></td>
</tr>
</tbody>
</table>

**Forms**  

`aloa = 25 (0x19)`

**Operand**  

... ⇒

**Stack**  

..., `objectref`

**Description**  The `index` is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The local variable at `index` must contain a reference. The `objectref` in the local variable at `index` is pushed onto the operand stack.

**Notes**  The `aloa` instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the `astore` instruction is intentional.

The `aloa` opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
**aload_<n>**  

**Operation** Load reference from local variable

**Format**  

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aload_&lt;n&gt;</strong></td>
</tr>
</tbody>
</table>

**Forms**  

- `aload_0 = 42 (0x2a)`  
- `aload_1 = 43 (0x2b)`  
- `aload_2 = 44 (0x2c)`  
- `aload_3 = 45 (0x2d)`

**Operand** ...

**Stack** ...

**Description** The `<n>` must be an index into the local variable array of the current frame (§3.6). The local variable at `<n>` must contain a reference. The `objectref` in the local variable at `index` is pushed onto the operand stack.

**Notes** An `aload_<n>` instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding `astore_<n>` instruction is intentional. Each of the `aload_<n>` instructions is the same as `aload` with an `index` of `<n>`, except that the operand `<n>` is implicit.
**anewarray**

**Operation**  Create new array of reference

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anewarray</td>
<td></td>
</tr>
<tr>
<td>indexbyte1</td>
<td></td>
</tr>
<tr>
<td>indexbyte2</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**  $anewarray = 189$ (0xbd)

**Operand**  $\ldots, count \Rightarrow$

**Stack**  $\ldots, arrayref$

**Description**  The $count$ must be of type int. It is popped off the operand stack. The $count$ represents the number of components of the array to be created. The unsigned $indexbyte1$ and $indexbyte2$ are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is $(indexbyte1 \ll 8) | indexbyte2$. The runtime constant pool item at that index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1). A new array with components of that type, of length $count$, is allocated from the garbage-collected heap, and a reference $arrayref$ to this new array object is pushed onto the operand stack. All components of the new array are initialized to null, the default value for reference types (§2.5.1).

**Linking Exceptions**  During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

**Runtime Exception**  Otherwise, if $count$ is less than zero, the $anewarray$ instruction throws a NegativeArraySizeException.

**Notes**  The $anewarray$ instruction is used to create a single dimension of an array of object references or part of a multidimensional array.
areturn

Operation  Return reference from method

Format  

Forms  areturn = 176 (0xb0)

Operand  ..., objectref ⇒
Stack  [empty]

Description  The objectref must be of type reference and must refer to an object of a type that is assignment compatible (§2.6.7) with the type represented by the return descriptor (§4.3.3) of the current method. If the current method is a synchronized method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a monitorexit instruction. If no exception is thrown, objectref is popped from the operand stack of the current frame (§3.6) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then reinstates the frame of the invoker and returns control to the invoker.

Runtime Exceptions  If the current method is a synchronized method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, areturn throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monitorenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in §8.13 and if the first of those rules is violated during invocation of the current method, then areturn throws an IllegalMonitorStateException.
**arraylength**  

**Operation** Get length of array

**Format**  

```markdown
arraylength
```

**Forms**  

arraylength = 190 (0xbe)

**Operand**  

.., arrayref =>

**Stack**  

.., length

**Description**  

The arrayref must be of type reference and must refer to an array. It is popped from the operand stack. The length of the array it refers to an array. It is popped from the operand stack. The length of the array it references is determined. That length is pushed onto the operand stack as an int.

**Runtime Exception**  

If the arrayref is null, the arraylength instruction throws a NullPointerException.
The *astore* instruction is used with an *objectref* of type `returnAddress` when implementing the `finally` clauses of the Java programming language (see Section 7.13, “Compiling `finally`”). The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

The *astore* opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
**astore_<n>**

**Operation**  
Store reference into local variable

**Format**  

<table>
<thead>
<tr>
<th>astore_&lt;n&gt;</th>
</tr>
</thead>
</table>

**Forms**

- astore_0 = 75 (0x4b)
- astore_1 = 76 (0x4c)
- astore_2 = 77 (0x4d)
- astore_3 = 78 (0x4e)

**Operand**  
..., objectref ⇒

**Stack**  
...

**Description**  
The <n> must be an index into the local variable array of the current frame (§3.6). The objectref on the top of the operand stack must be of type returnAddress or of type reference. It is popped from the operand stack, and the value of the local variable at <n> is set to objectref.

**Notes**  
An astore_<n> instruction is used with an objectref of type returnAddress when implementing the finally clauses of the Java programming language (see Section 7.13, “Compiling finally”). An aaload_<n> instruction cannot be used to load a value of type returnAddress from a local variable onto the operand stack. This asymmetry with the corresponding astore_<n> instruction is intentional.

Each of the astore_<n> instructions is the same as astore with an index of <n>, except that the operand <n> is implicit.
**athrow**

**Operation**
Throw exception or error

**Format**

```
athrow
```

**Forms**
`athrow = 191 (0xbf)`

**Operand Stack**
```
..., objectref ⇒
```

**Description**
The `objectref` must be of type `Reference` and must refer to an object that is an instance of class `Throwable` or of a subclass of `Throwable`. It is popped from the operand stack. The `objectref` is then thrown by searching the current method (§3.6) for the first exception handler that matches the class of `objectref`, as given by the algorithm in §3.10.

If an exception handler that matches `objectref` is found, it contains the location of the code intended to handle this exception. The pc register is reset to that location, the operand stack of the current frame is cleared, `objectref` is pushed back onto the operand stack, and execution continues.

If no matching exception handler is found in the current frame, that frame is popped. If the current frame represents an invocation of a synchronized method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a `monitorexit` instruction. Finally, the frame of its invoker is reinstated, if such a frame exists, and the `objectref` is rethrown. If no such frame exists, the current thread exits.

**Runtime Exceptions**
If `objectref` is `null`, `athrow` throws a `NullPointerException` instead of `objectref`.
**athrow (cont.)**

Otherwise, if the method of the current frame is a **synchronized** method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, _athrow_ throws an `IllegalMonitorStateException` instead of the object previously being thrown. This can happen, for example, if an abruptly completing **synchronized** method contains a `monitorexit` instruction, but no `monitorenter` instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in §8.13 and if the first of those rules is violated during invocation of the current method, then _athrow_ throws an `IllegalMonitorStateException` instead of the object previously being thrown.

**Notes**

The operand stack diagram for the _athrow_ instruction may be misleading: If a handler for this exception is matched in the current method, the _athrow_ instruction discards all the values on the operand stack, then pushes the thrown object onto the operand stack. However, if no handler is matched in the current method and the exception is thrown farther up the method invocation chain, then the operand stack of the method (if any) that handles the exception is cleared and `objectref` is pushed onto that empty operand stack. All intervening frames from the method that threw the exception up to, but not including, the method that handles the exception are discarded.
**baload**

**Operation**  
Load byte or boolean from array

**Format**

```
baload
```

**Forms**

`baload = 51` (0x33)

**Operand Stack**

`..., arrayref, index ⇒`,

`..., value`

**Description**  
The `arrayref` must be of type `reference` and must refer to an array whose components are of type `byte` or of type `boolean`. The `index` must be of type `int`. Both `arrayref` and `index` are popped from the operand stack. If the components of the array are of type `byte`, the component of the array at `index` is retrieved and sign-extended to an `int` value. The resulting `value` is pushed onto the operand stack.

**Runtime Exceptions**

If `arrayref` is `null`, `baload` throws a `NullPointerException`.

Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `baload` instruction throws an `ArrayIndexOutOfBoundsException`.

**Notes**  
The `baload` instruction is used to load values from both byte and boolean arrays. In Sun’s implementation of the Java virtual machine, boolean arrays (arrays of type `T_BOOLEAN`; see §3.2 and the description of the `newarray` instruction in this chapter) are implemented as arrays of 8-bit values. Other implementations may implement packed boolean arrays; the `baload` instruction of such implementations must be used to access those arrays.
The Java Virtual Machine Instruction Set

**bastore**

**Operation**
Store into byte or boolean array

**Format**

```
bastore
```

**Forms**

\[ bastore = 84 \ (0x54) \]

**Operand Stack**

\[ \ldots, \text{arrayref}, \text{index}, \text{value} \Rightarrow \ldots \]

**Description**
The `arrayref` must be of type reference and must refer to an array whose components are of type byte or of type boolean. The `index` and the `value` must both be of type `int`. The `arrayref`, `index`, and `value` are popped from the operand stack. If the components of the array are of type byte, the `int` `value` is truncated to a byte and stored as the component of the array indexed by `index`.

**Runtime Exceptions**
If `arrayref` is `null`, `bastore` throws a `NullPointerException`.
Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `bastore` instruction throws an `ArrayIndexOutOfBoundsException`.

**Notes**
The `bastore` instruction is used to store values into both byte and boolean arrays. In Sun's implementation of the Java virtual machine, boolean arrays (arrays of type `T_BOOLEAN`; see §3.2 and the description of the `newarray` instruction in this chapter) are implemented as arrays of 8-bit values. Other implementations may implement packed boolean arrays; in such implementations the `bastore` instruction must be able to store boolean values into packed boolean arrays as well as byte values into byte arrays.
**bipush**

**Operation**  Push byte

**Format**  

<table>
<thead>
<tr>
<th>bipush</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
</tr>
</tbody>
</table>

**Forms**  

$bipush = 16 \ (0x10)$

**Operand**  

... ⇒

**Stack**  

..., value

**Description**  The immediate byte is sign-extended to an int value. That value is pushed onto the operand stack.
**caload**

**Operation**  Load char from array

**Format**  

**Forms**  

**Operand**  

**Stack**  

**Description**  The arrayref must be of type reference and must refer to an array whose components are of type char. The index must be of type int. Both arrayref and index are popped from the operand stack. The component of the array at index is retrieved and zero-extended to an int value. That value is pushed onto the operand stack.

**Runtime Exceptions**  

If arrayref is null, caload throws a NullPointerException.

Otherwise, if index is not within the bounds of the array referenced by arrayref, the caload instruction throws an ArrayIndexOutOfBoundsException.
castore

Operation  Store into char array

Format  

Forms  castore = 85 (0x55)

Operand  ..., arrayref, index, value ⇒

Stack  ...

Description  The arrayref must be of type reference and must refer to an array whose components are of type char. The index and the value must both be of type int. The arrayref, index, and value are popped from the operand stack. The int value is truncated to a char and stored as the component of the array indexed by index.

Runtime Exceptions  If arrayref is null, castore throws a NullPointerException. Otherwise, if index is not within the bounds of the array referenced by arrayref, the castore instruction throws an ArrayIndexOutOfBoundsException.
**checkcast**

**Operation**  
Check whether object is of given type

**Format**

```
<table>
<thead>
<tr>
<th>checkcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>
```

**Forms**

`checkcast = 192 (0xc0)`

**Operand** 

..., `objectref` ⇒

**Stack** 

..., `objectref`

**Description**  
The `objectref` must be of type `reference`. The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1).

If `objectref` is null or can be cast to the resolved class, array, or interface type, the operand stack is unchanged; otherwise, the `checkcast` instruction throws a `ClassCastException`.

The following rules are used to determine whether an `objectref` that is not null can be cast to the resolved type: if `S` is the class of the object referred to by `objectref` and `T` is the resolved class, array, or interface type, `checkcast` determines whether `objectref` can be cast to type `T` as follows:

- If `S` is an ordinary (nonarray) class, then:
  - If `T` is a class type, then `S` must be the same class (§2.8.1) as `T`, or a subclass of `T`.
  - If `T` is an interface type, then `S` must implement (§2.13) interface `T`. 

DRAFT FOR REVIEW PURPOSES ONLY. DO NOT COPY WITHOUT WRITTEN PERMISSION FROM SUN MICROSYSTEMS, INC.
checkcast (cont.)

- If $S$ is an interface type, then:
  - If $T$ is a class type, then $T$ must be `Object` (§2.4.7).
  - If $T$ is an interface type, then $T$ must be the same interface as $S$ or a superinterface of $S$ (§2.13.2).
- If $S$ is a class representing the array type $SC[]$, that is, an array of components of type $SC$, then:
  - If $T$ is a class type, then $T$ must be `Object` (§2.4.7).
  - If $T$ is an array type $TC[]$, that is, an array of components of type $TC$, then one of the following must be true:
    - $TC$ and $SC$ are the same primitive type (§2.4.1).
    - $TC$ and $SC$ are reference types (§2.4.6), and type $SC$ can be cast to $TC$ by recursive application of these rules.
  - If $T$ is an interface type, $T$ must be one of the interfaces implemented by arrays (§2.15).

**Linking Exceptions**

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in Section 5.4.3.1 can be thrown.

**Runtime Exception**

Otherwise, if `objectref` cannot be cast to the resolved class, array, or interface type, the `checkcast` instruction throws a `ClassCastException`.

**Notes**

The `checkcast` instruction is very similar to the `instanceof` instruction. It differs in its treatment of null, its behavior when its test fails (`checkcast` throws an exception, `instanceof` pushes a result code), and its effect on the operand stack.
Operation  Convert double to float

Format  \[d2f\]

Forms  \[d2f = 144 \ (0x90)\]

Operand Stack  ..., value ⇒ ...

Stack  ..., result

Description  The value on the top of the operand stack must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3) resulting in value'. Then value' is converted to a float result using IEEE 754 round to nearest mode. The result is pushed onto the operand stack.

Where an \[d2f\] instruction is FP-strict (§3.8.2), the result of the conversion is always rounded to the nearest representable value in the float value set (§3.3.2).

Where an \[d2f\] instruction is not FP-strict, the result of the conversion may be taken from the float-extended-exponent value set (§3.3.2); it is not necessarily rounded to the nearest representable value in the float value set.

A finite value' too small to be represented as a float is converted to a zero of the same sign; a finite value' too large to be represented as a float is converted to an infinity of the same sign. A double NaN is converted to a float NaN.

Notes  The \[d2f\] instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of value' and may also lose precision.
**Operation**  Convert double to int

**Format**  

| d2i |

**Forms**  

\[ d2i = 142 \ (0x8e) \]

**Operand Stack**  

\[ ..., \text{value} \Rightarrow ..., \text{result} \]

**Description**  
The value on the top of the operand stack must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3) resulting in value'. Then value' is converted to an int. The result is pushed onto the operand stack:

- If the value' is NaN, the result of the conversion is an int 0.
- Otherwise, if the value' is not an infinity, it is rounded to an integer value \( V \), rounding towards zero using IEEE 754 round towards zero mode. If this integer value \( V \) can be represented as an int, then the result is the int value \( V \).
- Otherwise, either the value' must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type int, or the value' must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type int.

**Notes**  
The \( d2i \) instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of value' and may also lose precision.
**Operation**  Convert double to long

**Format**  

| \(d2l\) |

**Forms**  
\(d2l = 143 \ (0x8f)\)

**Operand Stack**  
\(\ldots, \text{value} \Rightarrow \ldots, \text{result}\)

**Description**  
The value on the top of the operand stack must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3) resulting in \(\text{value'}\). Then \(\text{value'}\) is converted to a long. The result is pushed onto the operand stack:

- If the \(\text{value'}\) is NaN, the result of the conversion is a long 0.
- Otherwise, if the \(\text{value'}\) is not an infinity, it is rounded to an integer value \(V\), rounding towards zero using IEEE 754 round towards zero mode. If this integer value \(V\) can be represented as a long, then the result is the long value \(V\).
- Otherwise, either the \(\text{value'}\) must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type long, or the \(\text{value'}\) must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type long.

**Notes**  
The \(d2l\) instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of \(\text{value'}\) and may also lose precision.
dadd

Operation  Add double

Format  

Forms  \( dadd = 99 \ (0x63) \)

Operand  \( \ldots, value1, value2 \Rightarrow \)

Stack  \( \ldots, result \)

Description  Both \( value1 \) and \( value2 \) must be of type double. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in \( value1' \) and \( value2' \). The double result is \( value1' + value2' \). The result is pushed onto the operand stack.

The result of a \( dadd \) instruction is governed by the rules of IEEE arithmetic:

- If either \( value1' \) or \( value2' \) is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a double, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a double, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a dadd instruction never throws a runtime exception.
**daload**

**Operation**  
Load double from array

**Format**  
```
daload
```

**Forms**  
```
daload = 49 (0x31)
```

**Operand Stack**  
```
..., arrayref, index ⇒
```

**Description**  
The `arrayref` must be of type `reference` and must refer to an array whose components are of type `double`. The `index` must be of type `int`. Both `arrayref` and `index` are popped from the operand stack. The `double` value in the component of the array at `index` is retrieved and pushed onto the operand stack.

**Runtime Exceptions**  
If `arrayref` is null, `daload` throws a `NullPointerException`. Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `daload` instruction throws an `ArrayIndexOutOfBoundsException`.

---

**daload**

---
**dastore**

**Operation**  Store into double array

**Format**  

**Forms**  \( \text{dastore} = 82 \ (0x52) \)

**Operand Stack**  \( \ldots, \text{arrayref}, \text{index}, \text{value} \Rightarrow \ldots \)

**Description**  The \( \text{arrayref} \) must be of type \text{reference} and must refer to an array whose components are of type \text{double}. The \( \text{index} \) must be of type \text{int}, and \( \text{value} \) must be of type \text{double}. The \( \text{arrayref}, \text{index}, \) and \( \text{value} \) are popped from the operand stack. The \text{double} \( \text{value} \) undergoes value set conversion (§3.8.3), resulting in \( \text{value'} \), which is stored as the component of the array indexed by \( \text{index} \).

**Runtime Exceptions**  
- If \( \text{arrayref} \) is \text{null}, \( \text{dastore} \) throws a \text{NullPointerException}.
- Otherwise, if \( \text{index} \) is not within the bounds of the array referenced by \( \text{arrayref} \), the \( \text{dastore} \) instruction throws an \text{ArrayIndexOutOfBoundsException}.
**Operation**  Compare double

**Format**  

\[
dcmp<op>
\]

**Forms**  
\[
dcmpg = 152 \ (0x98) \\
dcmpl = 151 \ (0x97)
\]

**Operand Stack**  
\[
..., \text{value1}, \text{value2} \Rightarrow \\
..., \text{result}
\]

**Description**  Both \text{value1} and \text{value2} must be of type \text{double}. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in \text{value1}' and \text{value2}'. A floating-point comparison is performed:

- If \text{value1}' is greater than \text{value2}', the \text{int} value 1 is pushed onto the operand stack.
- Otherwise, if \text{value1}' is equal to \text{value2}', the \text{int} value 0 is pushed onto the operand stack.
- Otherwise, if \text{value1}' is less than \text{value2}', the \text{int} value \(-1\) is pushed onto the operand stack.
- Otherwise, at least one of \text{value1}' or \text{value2}' is NaN. The \text{dcmpg} instruction pushes the \text{int} value 1 onto the operand stack and the \text{dcmpl} instruction pushes the \text{int} value \(-1\) onto the operand stack.

Floating-point comparison is performed in accordance with IEEE 754. All values other than NaN are ordered, with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.
The `dcmpg` and `dcmpl` instructions differ only in their treatment of a comparison involving NaN. NaN is unordered, so any double comparison fails if either or both of its operands are NaN. With both `dcmpg` and `dcmpl` available, any double comparison may be compiled to push the same result onto the operand stack whether the comparison fails on non-NaN values or fails because it encountered a NaN. For more information, see Section 7.5, “More Control Examples.”
### $dconst_{<d>}$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Push <code>double</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format</strong></td>
<td>$dconst_{&lt;d&gt;}$</td>
</tr>
</tbody>
</table>
| **Forms**       | $dconst_0 = 14$ (0xe)  
|                 | $dconst_1 = 15$ (0xf)  |
| **Operand**     | $\ldots \Rightarrow$ |
| **Stack**       | $\ldots$, $<d>$ |
| **Description** | Push the `double` constant $<d>$ (0.0 or 1.0) onto the operand stack. |
**ddiv**

**Operation**  Divide double

**Format**  

```

ddiv
```

**Forms**  

```
ddiv = 111 (0x6f)
```

**Operand Stack**  

```
..., value1, value2 \rightarrow
```

**Description**  

Both `value1` and `value2` must be of type `double`. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in `value1'` and `value2'`. The `double result` is `value1' / value2'`. The result is pushed onto the operand stack.

The result of a `ddiv` instruction is governed by the rules of IEEE arithmetic:

- If either `value1'` or `value2'` is NaN, the result is NaN.
- If neither `value1'` nor `value2'` is NaN, the sign of the result is positive if both values have the same sign, negative if the values have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- Division of a finite value by an infinity results in a signed zero, with the sign-producing rule just given.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero, with the sign-producing rule just given.
- Division of a nonzero finite value by a zero results in a signed infinity, with the sign-producing rule just given.
• In the remaining cases, where neither operand is an infinity, a zero, or NaN, the quotient is computed and rounded to the nearest double using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a double, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a double, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of a ddiv instruction never throws a runtime exception.
**dload**

**Operation**  Load double from local variable

**Format**

<table>
<thead>
<tr>
<th>dload</th>
<th>index</th>
</tr>
</thead>
</table>

**Forms**  
\[dload = 24 \ (0x18)\]

**Operand Stack**  
... ⇒ ...

... , value

**Description**  
The index is an unsigned byte. Both index and index + 1 must be indices into the local variable array of the current frame (§3.6). The local variable at index must contain a double. The value of the local variable at index is pushed onto the operand stack.

**Notes**  
The dload opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
**dload_<n>**

**Operation**  Load double from local variable

**Format**  

<table>
<thead>
<tr>
<th>dload_&lt;n&gt;</th>
</tr>
</thead>
</table>

**Forms**

- `dload_0 = 38 (0x26)`
- `dload_1 = 39 (0x27)`
- `dload_2 = 40 (0x28)`
- `dload_3 = 41 (0x29)`

**Operand Stack**

... ⇒ ...

, value

**Description**  Both `<n>` and `<n> + 1` must be indices into the local variable array of the current frame (§3.6). The local variable at `<n>` must contain a double. The value of the local variable at `<n>` is pushed onto the operand stack.

**Notes**  Each of the `dload_<n>` instructions is the same as `dload` with an index of `<n>`, except that the operand `<n>` is implicit.
Operation Multiply double

Format

Forms $dmul = 107 \ (0x6b)$

Operand Stack $\ldots, value1, value2 \Rightarrow$

Stack $\ldots, result$

Description Both $value1$ and $value2$ must be of type double. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in $value1'$ and $value2'$. The double result is $value1' \times value2'$. The result is pushed onto the operand stack.

The result of a $dmul$ instruction is governed by the rules of IEEE arithmetic:

- If either $value1'$ or $value2'$ is NaN, the result is NaN.

- If neither $value1'$ nor $value2'$ is NaN, the sign of the result is positive if both values have the same sign and negative if the values have different signs.

- Multiplication of an infinity by a zero results in NaN.

- Multiplication of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.

- In the remaining cases, where neither an infinity nor NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a double, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a double, we say the operation underflows; the result is then a zero of appropriate sign.
The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a `dmul` instruction never throws a runtime exception.
**dneg**

**Operation**  
Negate double

**Format**  
\[ \text{dneg} \]

**Forms**  
\[ \text{dneg} = 119 \ (0x77) \]

**Operand**  
\[ \ldots, \text{value} \Rightarrow \]

**Stack**  
\[ \ldots, \text{result} \]

**Description**  
The value must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value'. The double result is the arithmetic negation of value'. The result is pushed onto the operand stack.

For double values, negation is not the same as subtraction from zero. If \( x \) is +0.0, then 0.0–\( x \) equals +0.0, but −\( x \) equals −0.0. Unary minus merely inverts the sign of a double.

Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.
**Operation**  
Remainder double

**Format**  
\[
\begin{array}{|c|}
\hline
\text{drem} \\
\hline
\end{array}
\]

**Forms**  
drem = 115 (0x73)

**Operand**  
\[
\ldots, \text{value1}, \text{value2} \Rightarrow \\
\ldots, \text{result}
\]

**Description**  
Both value1 and value2 must be of type double. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in value1’ and value2’. The result is calculated and pushed onto the operand stack as a double.

The result of a drem instruction is not the same as that of the so-called remainder operation defined by IEEE 754. The IEEE 754 “remainder” operation computes the remainder from a rounding division, not a truncating division, and so its behavior is not analogous to that of the usual integer remainder operator. Instead, the Java virtual machine defines drem to behave in a manner analogous to that of the Java virtual machine integer remainder instructions (irem and lrem); this may be compared with the C library function fmod.

The result of a drem instruction is governed by these rules:

- If either value1’ or value2’ is NaN, the result is NaN.
- If neither value1’ nor value2’ is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity or the divisor is a zero or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
drem (cont.)

- If the dividend is a zero and the divisor is finite, the result equals the dividend.

- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder result from a dividend value1' and a divisor value2' is defined by the mathematical relation \( \text{result} = \text{value1}' - (\text{value2}' \times q) \), where \( q \) is an integer that is negative only if \( \text{value1}' / \text{value2}' \) is negative, and positive only if \( \text{value1}' / \text{value2}' \) is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of value1' and value2'.

Despite the fact that division by zero may occur, evaluation of a drem instruction never throws a runtime exception. Overflow, underflow, or loss of precision cannot occur.

Notes

The IEEE 754 remainder operation may be computed by the library routine Math.IEEEremainder.
**dreturn**

**Operation**
Return double from method

**Format**

```
dreturn
```

**Forms**
```
dreturn = 175 (0xaf)
```

**Operand Stack**
```
..., value ⇒
```

**Description**
The current method must have return type double. The value must be of type double. If the current method is a synchronized method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a monitorexit instruction. If no exception is thrown, value is popped from the operand stack of the current frame (§3.6) and undergoes value set conversion (§3.8.3), resulting in value'. The value' is pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

**Runtime Exceptions**
If the current method is a synchronized method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, dreturn throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monitorenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in §8.13 and if the first of those rules is violated during invocation of the current method, then dreturn throws an IllegalMonitorStateException.
**$dstore$**

**Operation**  Store double into local variable

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$dstore$</td>
<td></td>
</tr>
<tr>
<td>$index$</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**  $dstore = 57 (0x39)$

**Operand Stack**  $..., value \Rightarrow$

**Description**  The $index$ is an unsigned byte. Both $index$ and $index + 1$ must be indices into the local variable array of the current frame (§3.6). The $value$ on the top of the operand stack must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in $value'$. The local variables at $index$ and $index + 1$ are set to $value'$.

**Notes**  The $dstore$ opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
**dstore_<n>**

**Operation**  
Store double into local variable

**Format**

```
 dstore_<n>
```

**Forms**

- `dstore_0 = 71 (0x47)`
- `dstore_1 = 72 (0x48)`
- `dstore_2 = 73 (0x49)`
- `dstore_3 = 74 (0x4a)`

**Operand**

```
..., value ⇒
```

**Stack**

```
...
```

**Description**

Both `<n>` and `<n> + 1` must be indices into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type `double`. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in `value'`. The local variables at `<n>` and `<n> + 1` are set to `value'`.

**Notes**

Each of the `dstore_<n>` instructions is the same as `dstore` with an index of `<n>`, except that the operand `<n>` is implicit.
**dsub**

**Operation** Subtract double

**Format**

```
. .
```

**Forms**

\(dsub = 103 \text{ (0x67)}\)

**Operand Stack**

\(\ldots, value1, value2 \Rightarrow\)

**Description** Both value1 and value2 must be of type double. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in value1’ and value2’. The double result is value1’ − value2’. The result is pushed onto the operand stack.

For double subtraction, it is always the case that \(a−b\) produces the same result as \(a+(−b)\). However, for the dsub instruction, subtraction from zero is not the same as negation, because if \(x\) is +0.0, then 0.0−x equals +0.0, but −x equals −0.0.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a dsub instruction never throws a runtime exception.
**dup**

**Operation**  Duplicate the top operand stack value

**Format**  

```
dup
```

**Forms**  

`dup = 89 (0x59)`

**Operand**  

`..., value ⇒`

**Stack**  

`..., value, value`

**Description**  Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.

The `dup` instruction must not be used unless `value` is a value of a category 1 computational type (§3.11.1).
**dup_x1**

**Operation**  
Duplicate the top operand stack value and insert two values down

**Format**

```
dup_x1
```

**Forms**  
`dup_x1 = 90 (0x5a)`

**Operand**  
`..., value2, value1 ⇒`

**Stack**  
`..., value1, value2, value1`

**Description**  
Duplicate the top value on the operand stack and insert the duplicated value two values down in the operand stack.

The `dup_x1` instruction must not be used unless both `value1` and `value2` are values of a category 1 computational type (§3.11.1).
**dup_x2**

**Operation**
Duplicate the top operand stack value and insert two or three values down.

**Format**
```
dup_x2
```

**Forms**
```
dup_x2 = 91 (0x5b)
```

**Operand Stack**
Form 1:
```
..., value3, value2, value1 ⇒
..., value1, value3, value2, value1
```
where `value1`, `value2`, and `value3` are all values of a category 1 computational type (§3.11.1).

Form 2:
```
..., value2, value1 ⇒
..., value1, value2, value1
```
where `value1` is a value of a category 1 computational type and `value2` is a value of a category 2 computational type (§3.11.1).

**Description**
Duplicate the top value on the operand stack and insert the duplicated value two or three values down in the operand stack.
**Operation**
Duplicate the top one or two operand stack values

**Format**
```
dup2
```

**Forms**
`dup2 = 92 (0x5c)`

**Operand Stack**

Form 1:
```
..., value2, value1 ⇒
..., value2, value1, value2, value1
```
where both `value1` and `value2` are values of a category 1 computational type (§3.11.1).

Form 2:
```
..., value ⇒
..., value, value
```
where `value` is a value of a category 2 computational type (§3.11.1).

**Description**
Duplicate the top one or two values on the operand stack and push the duplicated value or values back onto the operand stack in the original order.
**dup2_x1**

**Operation**  Duplicate the top one or two operand stack values and insert two or three values down

**Format**  

| dup2_x1 |

**Forms**  

$dup2_x1 = 93 \ (0x5d)$

**Operand Stack**  

Form 1:

$\ldots, value3, value2, value1 \Rightarrow \ldots, value2, value1, value3, value2, value1$

where $value1$, $value2$, and $value3$ are all values of a category 1 computational type ($§3.11.1$).

Form 2:

$\ldots, value2, value1 \Rightarrow \ldots, value1, value2, value1$

where $value1$ is a value of a category 2 computational type and $value2$ is a value of a category 1 computational type ($§3.11.1$).

**Description**  Duplicate the top one or two values on the operand stack and insert the duplicated values, in the original order, one value beneath the original value or values in the operand stack.
**dup2_x2**

**Operation**
Duplicate the top one or two operand stack values and insert two, three, or four values down

**Format**
```
dup2_x2
```

**Forms**
\(\text{dup2}_x2 = 94 \text{ (0x5e)}\)

**Operands**

**Stack**

Form 1:
\[\ldots, \text{value4}, \text{value3}, \text{value2}, \text{value1} \Rightarrow \ldots, \text{value2}, \text{value1}, \text{value4}, \text{value3}, \text{value2}, \text{value1}\]

where \(\text{value1}, \text{value2}, \text{value3},\) and \(\text{value4}\) are all values of a category 1 computational type (§3.11.1).

Form 2:
\[\ldots, \text{value3}, \text{value2}, \text{value1} \Rightarrow \ldots, \text{value1}, \text{value3}, \text{value2}, \text{value1}\]

where \(\text{value1}\) is a value of a category 2 computational type and \(\text{value2}\) and \(\text{value3}\) are both values of a category 1 computational type (§3.11.1).

Form 3:
\[\ldots, \text{value3}, \text{value2}, \text{value1} \Rightarrow \ldots, \text{value2}, \text{value1}, \text{value3}, \text{value2}, \text{value1}\]

where \(\text{value1}\) and \(\text{value2}\) are both values of a category 1 computational type and \(\text{value3}\) is a value of a category 2 computational type (§3.11.1).

Form 4:
\[\ldots, \text{value2}, \text{value1} \Rightarrow \ldots, \text{value1}, \text{value2}, \text{value1}\]

where \(\text{value1}\) and \(\text{value2}\) are both values of a category 2 computational type (§3.11.1).
**dup2_x2 (cont.)**

**Description**  Duplicate the top one or two values on the operand stack and insert the duplicated values, in the original order, into the operand stack.
f2d

**Operation**  Convert float to double

**Format**  

| f2d |

**Forms**  

f2d = 141 (0x8d)

**Operand Stack**  

..., value ⇒ ...

**Description**  The value on the top of the operand stack must be of type float. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value’. Then value’ is converted to a double result. This result is pushed onto the operand stack.

**Notes**  Where an f2d instruction is FP-strict (§3.8.2) it performs a widening primitive conversion (§2.6.2). Because all values of the float value set (§3.3.2) are exactly representable by values of the double value set (§3.3.2), such a conversion is exact.

Where an f2d instruction is not FP-strict, the result of the conversion may be taken from the double-extended-exponent value set; it is not necessarily rounded to the nearest representable value in the double value set. However, if the operand value is taken from the float-extended-exponent value set and the target result is constrained to the double value set, rounding of value may be required.
**f2i**

**Operation** Convert float to int

**Format**

| f2i |

**Forms**

\[ f2i = 139 \ (0x8b) \]

**Operand Stack**

\[ \ldots, value \Rightarrow \ldots, result \]

**Description** The value on the top of the operand stack must be of type float. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value'. Then value' is converted to an int result. This result is pushed onto the operand stack:

- If the value' is NaN, the result of the conversion is an int 0.
- Otherwise, if the value' is not an infinity, it is rounded to an integer value V, rounding towards zero using IEEE 754 round towards zero mode. If this integer value V can be represented as an int, then the result is the int value V.
- Otherwise, either the value' must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type int, or the value' must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type int.

**Notes** The f2i instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of value' and may also lose precision.
**Operation** Convert *float* to *long*

**Format**

![f2l format]

**Forms** \( f2l = 140 \ (0x8c) \)

**Operand** \( \ldots, value \Rightarrow \)

**Stack** \( \ldots, result \)

**Description** The *value* on the top of the operand stack must be of type *float*. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in *value’*. Then *value’* is converted to a *long* *result*. This *result* is pushed onto the operand stack:

- If the *value’* is NaN, the *result* of the conversion is a *long* 0.
- Otherwise, if the *value’* is not an infinity, it is rounded to an integer value \( V \), rounding towards zero using IEEE 754 round towards zero mode. If this integer value \( V \) can be represented as a *long*, then the *result* is the *long* value \( V \).
- Otherwise, either the *value’* must be too small (a negative value of large magnitude or negative infinity), and the *result* is the smallest representable value of type *long*, or the *value’* must be too large (a positive value of large magnitude or positive infinity), and the *result* is the largest representable value of type *long*.

**Notes** The \( f2l \) instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of *value’* and may also lose precision.
**fadd**

**Operation**  Add float

**Format**  

<table>
<thead>
<tr>
<th>fadd</th>
</tr>
</thead>
</table>

**Forms**  

fadd = 98 (0x62)

**Operand Stack**  

..., value1, value2 ⇒ ...

..., result

**Description**  Both value1 and value2 must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in value1' and value2'. The float result is value1' + value2'. The result is pushed onto the operand stack.

The result of an fadd instruction is governed by the rules of IEEE arithmetic:

- If either value1' or value2' is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a float, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a float, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an fadd instruction never throws a runtime exception.
**faload**

**Operation**  
Load float from array

**Format**  

```
  faload
```

**Forms**  

\[ faload = 48 \ (0x30) \]

**Operand Stack**  

\[
    \ldots, \ arrayref, \ index \Rightarrow
\]

**Description**  
The arrayref must be of type reference and must refer to an array whose components are of type float. The index must be of type int. Both arrayref and index are popped from the operand stack. The float value in the component of the array at index is retrieved and pushed onto the operand stack.

**Runtime Exceptions**  
If arrayref is null, faload throws a NullPointerException.

Otherwise, if index is not within the bounds of the array referenced by arrayref, the faload instruction throws an ArrayIndexOutOfBoundsException.
The `fastore` instruction stores a `float` value into an array referenced by a `reference` object.

**Operation**
Store into `float` array

**Format**
```
fastore
```

**Forms**
`fastore = 81 (0x51)`

**Operand Stack**
```
..., arrayref, index, value ⇒
```

**Description**
The `arrayref` must be of type `reference` and must refer to an array whose components are of type `float`. The `index` must be of type `int`, and the `value` must be of type `float`. The `arrayref`, `index`, and `value` are popped from the operand stack. The `float` value undergoes value set conversion (§3.8.3), resulting in `value'`, and `value'` is stored as the component of the array indexed by `index`.

**Runtime Exceptions**
- If `arrayref` is null, `fastore` throws a `NullPointerException`.
- Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `fastore` instruction throws an `ArrayIndexOutOfBoundsException`.

```null
```
**fcmp<op>**

**Operation**  Compare float

**Format**

| fcmp<op> |

**Forms**

- `fcmpg` = 150 (0x96)
- `fcmpl` = 149 (0x95)

**Operand**  

... value1, value2 ⇒

**Stack**  

... result

**Description**  
Both value1 and value2 must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in value1' and value2'. A floating-point comparison is performed:

- If value1' is greater than value2', the int value 1 is pushed onto the operand stack.
- Otherwise, if value1' is equal to value2', the int value 0 is pushed onto the operand stack.
- Otherwise, if value1' is less than value2', the int value −1 is pushed onto the operand stack.
- Otherwise, at least one of value1' or value2' is NaN. The `fcmpg` instruction pushes the int value 1 onto the operand stack and the `fcmpl` instruction pushes the int value −1 onto the operand stack.

Floating-point comparison is performed in accordance with IEEE 754. All values other than NaN are ordered, with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.
The `fcmpg` and `fcmpl` instructions differ only in their treatment of a comparison involving NaN. NaN is unordered, so any `float` comparison fails if either or both of its operands are NaN. With both `fcmpg` and `fcmpl` available, any `float` comparison may be compiled to push the same `result` onto the operand stack whether the comparison fails on non-NaN values or fails because it encountered a NaN. For more information, see Section 7.5, “More Control Examples.”
**fconst_<f>**

**Operation**
Push float

**Format**
```
fconst_<f>
```

**Forms**
- `fconst_0 = 11 (0xb)`
- `fconst_1 = 12 (0xc)`
- `fconst_2 = 13 (0xd)`

**Operand**
...

**Stack**
...<f>

**Description**
Push the float constant `<f>` (0.0, 1.0, or 2.0) onto the operand stack.
Operation  Divide float

Format  \[ fdiv \]

Forms  \[ fdiv = 110 \ (0x6e) \]

Operand  \[ \ldots, \ value_1, \ value_2 \Rightarrow \]

Stack  \[ \ldots, \ result \]

Description  Both \( value_1 \) and \( value_2 \) must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in \( value_1' \) and \( value_2' \). The float result is \( value_1' / value_2' \). The result is pushed onto the operand stack.

The result of an \( fdiv \) instruction is governed by the rules of IEEE arithmetic:

- If either \( value_1' \) or \( value_2' \) is NaN, the result is NaN.
- If neither \( value_1' \) nor \( value_2' \) is NaN, the sign of the result is positive if both values have the same sign, negative if the values have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- Division of a finite value by an infinity results in a signed zero, with the sign-producing rule just given.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero, with the sign-producing rule just given.
- Division of a nonzero finite value by a zero results in a signed infinity, with the sign-producing rule just given.
In the remaining cases, where neither operand is an infinity, a zero, or NaN, the quotient is computed and rounded to the nearest float using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a float, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a float, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of an fdiv instruction never throws a runtime exception.
**fload**

**Operation**  Load float from local variable

**Format**

|     |  
|-----|---
| **fload** |  
| **index** |  

**Forms**  

fload = 23 (0x17)

**Operand**

... ⇒

**Stack**

..., value

**Description**  The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The local variable at index must contain a float. The value of the local variable at index is pushed onto the operand stack.

**Notes**  The fload opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
\textit{fload}_{<n>}

**Operation**  Load \texttt{float} from local variable

**Format**  \[ fload_{<n>} \]

**Forms**  
- \texttt{fload}_0 = 34 (0x22)
- \texttt{fload}_1 = 35 (0x23)
- \texttt{fload}_2 = 36 (0x24)
- \texttt{fload}_3 = 37 (0x25)

**Operand**  \( \ldots \Rightarrow \)

**Stack**  \( \ldots, \text{value} \)

**Description**  The \(<n>\) must be an index into the local variable array of the current frame (§3.6). The local variable at \(<n>\) must contain a \texttt{float}. The \textit{value} of the local variable at \(<n>\) is pushed onto the operand stack.

**Notes**  Each of the \textit{fload}_{<n>} instructions is the same as \textit{fload} with an index of \(<n>\), except that the operand \(<n>\) is implicit.
**fmul**

**Operation**  Multiply float

**Format**  

**Forms**  \( fmul = 106 \, (0x6a) \)

**Operand**  \( \ldots, \text{value1, value2} \Rightarrow \)

**Stack**  \( \ldots, \text{result} \)

**Description**  Both \( \text{value1} \) and \( \text{value2} \) must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in \( \text{value1}' \) and \( \text{value2}' \). The float result is \( \text{value1}' \ast \text{value2}' \). The result is pushed onto the operand stack.

The result of an \( fmul \) instruction is governed by the rules of IEEE arithmetic:

- If either \( \text{value1}' \) or \( \text{value2}' \) is NaN, the result is NaN.
- If neither \( \text{value1}' \) nor \( \text{value2}' \) is NaN, the sign of the result is positive if both values have the same sign, and negative if the values have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither an infinity nor NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a float, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a float, we say the operation underflows; the result is then a zero of appropriate sign.
The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an \textit{fmul} instruction never throws a runtime exception.
**fneg**

**Operation**  Negate float

**Format**  

| fneg |

**Forms**  

\[ f\text{neg} = 118 \ (0x76) \]

**Operand Stack**  

\[ \ldots, \text{value} \Rightarrow \ldots, \text{result} \]

**Description**  The value must be of type float. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value’. The float result is the arithmetic negation of value’. This result is pushed onto the operand stack.

For float values, negation is not the same as subtraction from zero. If \( x \) is +0.0, then 0.0−x equals +0.0, but −x equals −0.0. Unary minus merely inverts the sign of a float.

Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.
frem

**Operation**  
Remainder float

**Format**  
frem

**Forms**  
frem = 114 (0x72)

**Operand**  
..., value1, value2 ⇒

**Stack**  
..., result

**Description**  
Both value1 and value2 must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in value1' and value2'. The result is calculated and pushed onto the operand stack as a float.

The result of an frem instruction is not the same as that of the so-called remainder operation defined by IEEE 754. The IEEE 754 “remainder” operation computes the remainder from a rounding division, not a truncating division, and so its behavior is not analogous to that of the usual integer remainder operator. Instead, the Java virtual machine defines frem to behave in a manner analogous to that of the Java virtual machine integer remainder instructions (irem and lrem); this may be compared with the C library function fmod.

The result of an frem instruction is governed by these rules:

- If either value1' or value2' is NaN, the result is NaN.
- If neither value1' nor value2' is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity or the divisor is a zero or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
frem (cont.) frem (cont.)

- If the dividend is a zero and the divisor is finite, the result equals the dividend.

- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder result from a dividend value1' and a divisor value2' is defined by the mathematical relation result = value1' \(−(value2' \ast q)\), where q is an integer that is negative only if value1' / value2' is negative and positive only if value1' / value2' is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of value1' and value2'.

Despite the fact that division by zero may occur, evaluation of an frem instruction never throws a runtime exception. Overflow, underflow, or loss of precision cannot occur.

Notes The IEEE 754 remainder operation may be computed by the library routine Math.IEEEremainder.
**freturn**

**Operation**
Return float from method

**Format**

**Forms**
\[f\text{return} = 174 \ (0x\text{ae})\]

**Operand Stack**
\[\ldots, \text{value} \Rightarrow\]

**Description**
The current method must have return type float. The value must be of type float. If the current method is a synchronized method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a monitorexit instruction. If no exception is thrown, value is popped from the operand stack of the current frame (§3.6) and undergoes value set conversion (§3.8.3), resulting in value'. The value' is pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

**Runtime Exceptions**
If the current method is a synchronized method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, freturn throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monitorenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in §8.13 and if the first of those rules is violated during invocation of the current method, then freturn throws an IllegalMonitorStateException.
**fstore**

**Operation**  Store float into local variable

**Format**

<table>
<thead>
<tr>
<th></th>
<th>fstore</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>index</td>
</tr>
</tbody>
</table>

**Forms**  

fstore = 56 (0x38)

**Operand**  

..., value ⇒

**Stack**  

...

**Description**  
The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The value on the top of the operand stack must be of type float. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value'. The value of the local variable at index is set to value'.

**Notes**  
The fstore opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
**Operation**  
Store `float` into local variable

**Format**

```
fstore_<n>
```

**Forms**

- `fstore_0 = 67 (0x43)`
- `fstore_1 = 68 (0x44)`
- `fstore_2 = 69 (0x45)`
- `fstore_3 = 70 (0x46)`

**Operand**  
...

**Stack**  
...

**Description**
The `<n>` must be an index into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type `float`. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in `value'`. The value of the local variable at `<n>` is set to `value'`.

**Notes**
Each of the `fstore_<n>` is the same as `fstore` with an *index* of `<n>`, except that the operand `<n>` is implicit.
**fsub**

**Operation** Subtract float

**Format**

```
```

**Forms** 

$fsub = 102$ (0x66)

**Operand** 

\[..., \text{value1, value2} \Rightarrow\]

**Stack** 

\[..., \text{result}\]

**Description** Both value1 and value2 must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in value1' and value2'. The float result is value1' − value2'. The result is pushed onto the operand stack.

For float subtraction, it is always the case that a−b produces the same result as a+(−b). However, for the fsub instruction, subtraction from zero is not the same as negation, because if x is +0.0, then 0.0−x equals +0.0, but −x equals −0.0.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an fsub instruction never throws a runtime exception.
### `getfield`

**Operation**  
Fetch field from object

**Format**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getfield</code></td>
<td></td>
</tr>
<tr>
<td><code>indexbyte1</code></td>
<td></td>
</tr>
<tr>
<td><code>indexbyte2</code></td>
<td></td>
</tr>
</tbody>
</table>

**Forms**  
`getfield = 180 (0xb4)`

**Operand Stack**

1. ..., `objectref` ⇒
2. ..., `value`

**Description**  
The `objectref`, which must be of type `reference`, is popped from the operand stack. The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.2). The `value` of the referenced field in `objectref` is fetched and pushed onto the operand stack.

The class of `objectref` must not be an array. If the field is protected (§4.6), and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of `objectref` must be either the current class or a subclass of the current class.

**Linking Exceptions**  
During resolution of the symbolic reference to the field, any of the errors pertaining to field resolution documented in Section 5.4.3.2 can be thrown.

Otherwise, if the resolved field is a static field, `getfield` throws an `IncompatibleClassChangeError`.

**Other**

1. ..., `objectref` ⇒
2. ..., `value`
getfield (cont.)

Runtime Exception

Otherwise, if objectref is null, the getfield instruction throws a NullPointerException.

Notes

The getfield instruction cannot be used to access the length field of an array. The arraylength instruction is used instead.
**getstatic**

**Operation**  Get static field from class

**Format**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>getstatic</td>
<td>indexbyte1</td>
</tr>
<tr>
<td></td>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms**  getstatic = 178 (0xb2)

**Operand**  ..., ⇒

**Stack**  ..., value

**Description**  The unsigned indexbyte1 and indexbyte2 are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is (indexbyte1 << 8) | indexbyte2. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved (§5.4.3.2).

On successful resolution of the field, the class or interface that declared the resolved field is initialized (§5.5) if that class or interface has not already been initialized.

The value of the class or interface field is fetched and pushed onto the operand stack.

**Linking Exceptions**  During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution documented in Section 5.4.3.2 can be thrown.

Otherwise, if the resolved field is not a static (class) field or an interface field, getstatic throws an IncompatibleClassChangeError.
getstatic (cont.)

Runtime Exception

Otherwise, if execution of this getstatic instruction causes initialization of the referenced class or interface, getstatic may throw an Error as detailed in Section 2.17.5.
goto

Operation  Branch always

Format

| goto  
| branchbyte1 
| branchbyte2 |

Forms  \( goto = 167 \) (0xa7)

Operand Stack  No change

Description  The unsigned bytes \( \text{branchbyte1} \) and \( \text{branchbyte2} \) are used to construct a signed 16-bit \( \text{branchoffset} \), where \( \text{branchoffset} \) is \( (\text{branchbyte1} \ll 8) | \text{branchbyte2} \). Execution proceeds at that offset from the address of the opcode of this \( goto \) instruction. The target address must be that of an opcode of an instruction within the method that contains this \( goto \) instruction.
**goto_w**

**Operation**  
Branch always (wide index)

**Format**

<table>
<thead>
<tr>
<th>goto_w</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
<tr>
<td>branchbyte3</td>
</tr>
<tr>
<td>branchbyte4</td>
</tr>
</tbody>
</table>

**Forms**  
goto_w = 200 (0xc8)

**Operand**  
No change

**Stack**

**Description**  
The unsigned bytes `branchbyte1`, `branchbyte2`, `branchbyte3`, and `branchbyte4` are used to construct a signed 32-bit `branchoffset`, where `branchoffset` is `(branchbyte1 << 24) | (branchbyte2 << 16) | (branchbyte3 << 8) | branchbyte4`. Execution proceeds at that offset from the address of the opcode of this `goto_w` instruction. The target address must be that of an opcode of an instruction within the method that contains this `goto_w` instruction.

**Notes**  
Although the `goto_w` instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes (§4.10). This limit may be raised in a future release of the Java virtual machine.
### i2b

**Operation** Convert int to byte

**Format**

```
i2b
```

**Forms**

\( i2b = 145 \) (0x91)

**Operand Stack** ...., value ⇒

**Stack** ...., result

**Description** The value on the top of the operand stack must be of type int. It is popped from the operand stack, truncated to a byte, then sign-extended to an int result. That result is pushed onto the operand stack.

**Notes** The i2b instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of value. The result may also not have the same sign as value.
**Operation**  Convert int to char

**Format**  

```
  i2c
```

**Forms**  

\(i2c = 146 \ (0x92)\)

**Operand Stack**  

\(\ldots, \text{value} \Rightarrow \ldots, \text{result}\)

**Description**  The value on the top of the operand stack must be of type int. It is popped from the operand stack, truncated to char, then zero-extended to an int result. That result is pushed onto the operand stack.

**Notes**  The \(i2c\) instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of value. The result (which is always positive) may also not have the same sign as value.
\textit{i2d} \hfill \textit{i2d}

\begin{itemize}
  \item \textbf{Operation} \hfill Convert \textit{int} to \textit{double}
  \item \textbf{Format} \hfill \textit{i2d}
  \item \textbf{Forms} \hfill \textit{i2d} = 135 (0x87)
  \item \textbf{Operand Stack} \hfill \ldots, \textit{value} \Rightarrow
  \item \textbf{Stack} \hfill \ldots, \textit{result}
  \item \textbf{Description} \hfill The \textit{value} on the top of the operand stack must be of type \textit{int}. It is popped from the operand stack and converted to a \textit{double} \textit{result}. The \textit{result} is pushed onto the operand stack.
  \item \textbf{Notes} \hfill The \textit{i2d} instruction performs a widening primitive conversion (§2.6.2). Because all values of type \textit{int} are exactly representable by type \textit{double}, the conversion is exact.
\end{itemize}
Operation: Convert int to float

Format: \texttt{i2f}

Forms: \texttt{i2f} = 134 (0x86)

Operand: \ldots, \texttt{value} \Rightarrow

Stack: \ldots, \texttt{result}

Description: The \texttt{value} on the top of the operand stack must be of type \texttt{int}. It is popped from the operand stack and converted to the \texttt{float} \texttt{result} using IEEE 754 round to nearest mode. The \texttt{result} is pushed onto the operand stack.

Notes: The \texttt{i2f} instruction performs a widening primitive conversion (§2.6.2), but may result in a loss of precision because values of type \texttt{float} have only 24 significand bits.
\textbf{Operation} \hspace{1em} Convert \texttt{int} to \texttt{long}

\textbf{Format} \hspace{1em} \texttt{i2l}

\textbf{Forms} \hspace{1em} \texttt{i2l} = 133 (0x85)

\textbf{Operand} \hspace{1em} \ldots, \texttt{value} \Rightarrow \ldots

\textbf{Stack} \hspace{1em} \ldots, \texttt{result}

\textbf{Description} \hspace{1em} The \texttt{value} on the top of the operand stack must be of type \texttt{int}. It is popped from the operand stack and sign-extended to a \texttt{long} \texttt{result}. That \texttt{result} is pushed onto the operand stack.

\textbf{Notes} \hspace{1em} The \texttt{i2l} instruction performs a widening primitive conversion (§2.6.2). Because all values of type \texttt{int} are exactly representable by type \texttt{long}, the conversion is exact.
### i2s

**Operation**  Convert int to short

**Format**  

| i2s |

**Forms**  

i2s = 147 (0x93)

**Operand Stack**  

... \(\text{value} \Rightarrow\) ...

**Description**  

The value on the top of the operand stack must be of type int. It is popped from the operand stack, truncated to a short, then sign-extended to an int result. That result is pushed onto the operand stack.

**Notes**  

The i2s instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of value. The result may also not have the same sign as value.
iadd

**Operation**  Add int

**Format**  

| iadd |

**Forms**  iadd = 96 (0x60)

**Operand Stack**  

..., value1, value2 ⇒

..., result

**Description**  Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 + value2. The result is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type int. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an iadd instruction never throws a runtime exception.
**iaload**

**Operation**  Load int from array

**Format**

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>iaload</strong></td>
</tr>
</tbody>
</table>

**Forms**  

<table>
<thead>
<tr>
<th>Forms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>iaload</strong> = 46 (0x2e)</td>
</tr>
</tbody>
</table>

**Operand**  

<table>
<thead>
<tr>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>…. arrayref, index ⇒</td>
</tr>
</tbody>
</table>

**Stack**  

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>…. value</td>
</tr>
</tbody>
</table>

**Description**  

The arrayref must be of type reference and must refer to an array whose components are of type int. The index must be of type int. Both arrayref and index are popped from the operand stack. The int value in the component of the array at index is retrieved and pushed onto the operand stack.

**Runtime Exceptions**  

If arrayref is null, iaload throws a NullPointerException.

Otherwise, if index is not within the bounds of the array referenced by arrayref, the iaload instruction throws an ArrayIndexOutOfBoundsException.
**Operation**  
Boolean AND int

**Format**  
```
  iand
```

**Forms**  
```
iand = 126 (0x7e)
```

**Operand**  
```
..., value1, value2 ⇒
```

**Stack**  
```
..., result
```

**Description**  
Both `value1` and `value2` must be of type `int`. They are popped from the operand stack. An `int` result is calculated by taking the bitwise AND (conjunction) of `value1` and `value2`. The `result` is pushed onto the operand stack.
**iastore**

**Operation**  
Store into int array

**Format**  
```
 iastore
```

**Forms**  
*iastore* = 79 (0x4f)

**Operand Stack**  
```
..., arrayref, index, value =>
```

**Description**  
The *arrayref* must be of type reference and must refer to an array whose components are of type int. Both *index* and *value* must be of type int. The *arrayref*, *index*, and *value* are popped from the operand stack. The int *value* is stored as the component of the array indexed by *index*.

**Runtime Exceptions**  
If *arrayref* is null, *iastore* throws a NullPointerException.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *iastore* instruction throws an ArrayIndexOutOfBoundsException.
**iconst_<i>**

**Operation**
Push int constant

**Format**

<table>
<thead>
<tr>
<th>Format</th>
<th>iconst_&lt;i&gt;</th>
</tr>
</thead>
</table>

**Forms**
iconst_m1 = 2 (0x2)  
iconst_0 = 3 (0x3)  
iconst_1 = 4 (0x4)  
iconst_2 = 5 (0x5)  
iconst_3 = 6 (0x6)  
iconst_4 = 7 (0x7)  
iconst_5 = 8 (0x8)

**Operand**
...

**Stack**
..., <i>

**Description**
Push the int constant <i> (−1, 0, 1, 2, 3, 4 or 5) onto the operand stack.

**Notes**
Each of this family of instructions is equivalent to bipush <i> for the respective value of <i>, except that the operand <i> is implicit.
**idiv**

**Operation**  Divide int

**Format**  

```
    idiv
```

**Forms**  

```
idiv = 108 (0x6c)
```

**Operand**  

```
..., value1, value2 ⇒
```

**Stack**  

```
..., result
```

**Description**  Both `value1` and `value2` must be of type int. The values are popped from the operand stack. The int `result` is the value of the Java programming language expression `value1 / value2`. The `result` is pushed onto the operand stack.

An int division rounds towards 0; that is, the quotient produced for int values in `n/d` is an int value `q` whose magnitude is as large as possible while satisfying `|d \cdot q| \leq |n|`. Moreover, `q` is positive when `|n| \geq |d|` and `n` and `d` have the same sign, but `q` is negative when `|n| \geq |d|` and `n` and `d` have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for the int type, and the divisor is −1, then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case.

**Runtime Exception**  If the value of the divisor in an int division is 0, `idiv` throws an `ArithmeticException`. 
### if\_acmp<cond> \hspace{1cm} if\_acmp<cond>

**Operation**  
Branch if reference comparison succeeds

**Format**

<table>
<thead>
<tr>
<th>if_acmp&lt;cond&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

**Forms**

if\_acmpeq = 165 (0xa5)  
if\_acmpne = 166 (0xa6)

**Operand**  
\(\ldots, \text{value1}, \text{value2} \Rightarrow\)

**Stack**  
\(\ldots\)

**Description**

Both value1 and value2 must be of type reference. They are both popped from the operand stack and compared. The results of the comparison are as follows:

- **eq** succeeds if and only if value1 = value2
- **ne** succeeds if and only if value1 ≠ value2

If the comparison succeeds, the unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be \((\text{branchbyte1} \ll 8) | \text{branchbyte2}\). Execution then proceeds at that offset from the address of the opcode of this if\_acmp<cond> instruction. The target address must be that of an opcode of an instruction within the method that contains this if\_acmp<cond> instruction.

Otherwise, if the comparison fails, execution proceeds at the address of the instruction following this if\_acmp<cond> instruction.
ificmp<cond>

Operation  Branch if int comparison succeeds

Format

<table>
<thead>
<tr>
<th>ificmp&lt;cond&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

Forms

if_icmpeq = 159 (0x9f)
if_icmpne = 160 (0xa0)
if_icmplt = 161 (0xa1)
if_icmpge = 162 (0xa2)
if_icmpgt = 163 (0xa3)
if_icmple = 164 (0xa4)

Operand  ...., value1, value2 ⇒
Stack  ...

Description  Both value1 and value2 must be of type int. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparison are as follows:

- eq  succeeds if and only if value1 = value2
- ne  succeeds if and only if value1 ≠ value2
- lt  succeeds if and only if value1 < value2
- le  succeeds if and only if value1 ≤ value2
- gt  succeeds if and only if value1 > value2
- ge  succeeds if and only if value1 ≥ value2
If the comparison succeeds, the unsigned \textit{branchbyte1} and \textit{branchbyte2} are used to construct a signed 16-bit offset, where the offset is calculated to be \((\text{branchbyte1} \ll 8) \mid \text{branchbyte2}\). Execution then proceeds at that offset from the address of the opcode of this \textit{if_icmp<cond>} instruction. The target address must be that of an opcode of an instruction within the method that contains this \textit{if_icmp<cond>} instruction.

Otherwise, execution proceeds at the address of the instruction following this \textit{if_icmp<cond>} instruction.
Operation  Branch if int comparison with zero succeeds

Format  

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>if&lt;cond&gt;</td>
<td></td>
</tr>
<tr>
<td>branchbyte1</td>
<td></td>
</tr>
<tr>
<td>branchbyte2</td>
<td></td>
</tr>
</tbody>
</table>

Forms  
ifeq = 153 (0x99)  
ifne = 154 (0x9a)  
iflt = 155 (0x9b)  
ifge = 156 (0x9c)  
ifgt = 157 (0x9d)  
ifle = 158 (0x9e)

Operand  …. value ⇒
Stack  …. 

Description  The value must be of type int. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

• eq  succeeds if and only if value = 0
• ne  succeeds if and only if value ≠ 0
• lt  succeeds if and only if value < 0
• le  succeeds if and only if value ≤ 0
• gt  succeeds if and only if value > 0
• ge  succeeds if and only if value ≥ 0
If the comparison succeeds, the unsigned `branchbyte1` and `branchbyte2` are used to construct a signed 16-bit offset, where the offset is calculated to be `(branchbyte1 << 8) | branchbyte2`. Execution then proceeds at that offset from the address of the opcode of this `if<cond>` instruction. The target address must be that of an opcode of an instruction within the method that contains this `if<cond>` instruction.

Otherwise, execution proceeds at the address of the instruction following this `if<cond>` instruction.
ifnonnull

Operation  Branch if reference not null

Format

<table>
<thead>
<tr>
<th>ifnonnull</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

Forms  ifnonnull = 199 (0xc7)

Operand  ..., value ⇒

Stack    ...

Description  The value must be of type reference. It is popped from the operand stack. If value is not null, the unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be (branchbyte1 << 8) | branchbyte2. Execution then proceeds at that offset from the address of the opcode of this ifnonnull instruction. The target address must be that of an opcode of an instruction within the method that contains this ifnonnull instruction.

Otherwise, execution proceeds at the address of the instruction following this ifnonnull instruction.
ifnull

Operation Branch if reference is null

Format

<table>
<thead>
<tr>
<th>ifnull</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

Forms ifnull = 198 (0xc6)

Operand ... value ⇒

Stack ...

Description The value must of type reference. It is popped from the operand stack. If value is null, the unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be (branchbyte1 << 8) | branchbyte2. Execution then proceeds at that offset from the address of the opcode of this ifnull instruction. The target address must be that of an opcode of an instruction within the method that contains this ifnull instruction.

Otherwise, execution proceeds at the address of the instruction following this ifnull instruction.
**iinc**

**Operation**
Increment local variable by constant

**Format**

<table>
<thead>
<tr>
<th>iinc</th>
<th>index</th>
<th>const</th>
</tr>
</thead>
</table>

**Forms**

iinc = 132 (0x84)

**Operand Stack**
No change

**Description**
The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The *const* is an immediate signed byte. The local variable at *index* must contain an int. The value *const* is first sign-extended to an int, and then the local variable at *index* is incremented by that amount.

**Notes**
The *iinc* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index and to increment it by a two-byte immediate value.
**Iload**

**Operation**  Load int from local variable

**Format**

```
  | iload  |
  | index  |
```

**Forms**

`iload = 21 (0x15)`

**Operand Stack**

`... ⇒ ...
..., value`

**Description**

The `index` is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The local variable at `index` must contain an int. The value of the local variable at `index` is pushed onto the operand stack.

**Notes**

The `iload` opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
**iload_<n>**

**Operation**  
Load int from local variable

**Format**  

```
format iload_<n>
```

**Forms**  

- `iload_0 = 26` (0x1a)
- `iload_1 = 27` (0x1b)
- `iload_2 = 28` (0x1c)
- `iload_3 = 29` (0x1d)

**Operand Stack**  

```
... ⇒ ...
```

**Description**  
The `<n>` must be an index into the local variable array of the current frame (§3.6). The local variable at `<n>` must contain an int. The value of the local variable at `<n>` is pushed onto the operand stack.

**Notes**  
Each of the `iload_<n>` instructions is the same as `iload` with an index of `<n>`, except that the operand `<n>` is implicit.
**imul**

**Operation** Multiply int

**Format**  
```
imul
```

**Forms**  
`imul = 104 (0x68)`

**Operand**  
`..., value1, value2 ⇒`

**Stack**  
`..., result`

**Description** Both `value1` and `value2` must be of type `int`. The values are popped from the operand stack. The int `result` is `value1 * value2`. The `result` is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an `imul` instruction never throws a runtime exception.
**Operation**  
Negate int

**Format**  
```
ing
```  

**Forms**  
```
ing = 116 (0x74)
```  

**Operand**  
```
..., value ⇒
```  

**Stack**  
```
..., result
```  

**Description**  
The value must be of type int. It is popped from the operand stack. The int result is the arithmetic negation of value, −value. The result is pushed onto the operand stack.

For int values, negation is the same as subtraction from zero. Because the Java virtual machine uses two’s-complement representation for integers and the range of two’s-complement values is not symmetric, the negation of the maximum negative int results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all int values x, −x equals (−x) + 1.
**instanceof**

**Operation**
Determine if object is of given type

**Format**

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instanceof</code></td>
</tr>
<tr>
<td><code>indexbyte1</code></td>
</tr>
<tr>
<td><code>indexbyte2</code></td>
</tr>
</tbody>
</table>

**Forms**
`instanceof = 193 (0xc1)`

**Operand**
`..., objectref ⇒`

**Stack**
`..., result`

**Description**
The `objectref`, which must be of type `reference`, is popped from the operand stack. The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1).

If `objectref` is not null and is an instance of the resolved class or array or implements the resolved interface, the `instanceof` instruction pushes an int `result` of 1 as an int on the operand stack. Otherwise, it pushes an int `result` of 0.

The following rules are used to determine whether an `objectref` that is not null is an instance of the resolved type: If `S` is the class of the object referred to by `objectref` and `T` is the resolved class, array, or interface type, `instanceof` determines whether `objectref` is an instance of `T` as follows:

- If `S` is an ordinary (nonarray) class, then:
  - If `T` is a class type, then `S` must be the same class (§2.8.1) as `T` or a subclass of `T`.
  - If `T` is an interface type, then `S` must implement (§2.13) interface `T`. 
**instanceof** (cont.)

- If \( S \) is an interface type, then:
  - If \( T \) is a class type, then \( T \) must be \texttt{Object} (§2.4.7).
  - If \( T \) is an interface type, then \( T \) must be the same interface as \( S \), or a superinterface of \( S \) (§2.13.2).

- If \( S \) is a class representing the array type \( SC[] \), that is, an array of components of type \( SC \), then:
  - If \( T \) is a class type, then \( T \) must be \texttt{Object} (§2.4.7).
  - If \( T \) is an array type \( TC[] \), that is, an array of components of type \( TC \), then one of the following must be true:
    - \( TC \) and \( SC \) are the same primitive type (§2.4.1).
    - \( TC \) and \( SC \) are reference types (§2.4.6), and type \( SC \) can be cast to \( TC \) by these runtime rules.
  - If \( T \) is an interface type, \( T \) must be one of the interfaces implemented by arrays (§2.15).

**Linking Exceptions**

During resolution of symbolic reference to the class, array, or interface type, any of the exceptions documented in Section 5.4.3.1 can be thrown.

**Notes**

The `instanceof` instruction is very similar to the `checkcast` instruction. It differs in its treatment of `null`, its behavior when its test fails (`checkcast` throws an exception, `instanceof` pushes a result code), and its effect on the operand stack.
**invokedynamic**

**Operation** Invoke instance method; resolve and dispatch based on class

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokedynamic</td>
</tr>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms** $\text{invokedynamic} = 186 (0xba)$

**Operand** \[..., objectref, [arg1, [arg2 ...]]\] \(\Rightarrow\)

**Stack** \(\ldots\)

**Description** The unsigned $\text{indexbyte1}$ and $\text{indexbyte2}$ are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is $(\text{indexbyte1} << 8) | \text{indexbyte2}$. The runtime constant pool item at that index must be a CONSTANT_NameAndType_info (§4.4.6), which gives the name and descriptor (§4.3.3) of a method. The referenced method name must not name an instance initialization method (§3.9) or class or interface initialization method (§3.9).

The $\text{objectref}$ must be followed on the operand stack by $\text{nargs}$ argument values of reference type, where the number and order of the values must be consistent with the referenced descriptor.

Let $C$ be the class of $\text{objectref}$. The actual method to be invoked is selected by the following lookup procedure:

- If $C$ contains a declaration for an instance method $M$ with the same name and descriptor as the referenced method, then $M$ is the method to be invoked, and the lookup procedure terminates.
Otherwise, if \( C \) has a superclass, this same lookup procedure is performed recursively using the direct superclass of \( C \); the method to be invoked is the result of the recursive invocation of this lookup procedure.

Otherwise, if no method matching the referenced name and descriptor is selected, *invokedynamic* invokes the method named `handleMethodInvocationError` with descriptor `([Ljava/lang/Object;)Ljava/lang/Object;` on `objectref`, with an argument that is an object array whose zeroth element is the name of the referenced method, whose first element is the descriptor of the resolved method, whose second element is an instance of the class `NoSuchMethodError` and whose subsequent elements are the original arguments. Then result of the call to `handleMethodInvocationError` is pushed onto the operand stack of the invoker.

If the selected method is abstract, *invokedynamic* invokes the method named `handleMethodInvocationError` with descriptor `([Ljava/lang/Object;)Ljava/lang/Object;` on `objectref`, with an argument that is an object array whose zeroth element is the name of the referenced method, whose first element is the descriptor of the resolved method, whose second element is an instance of the class `AbstractMethodError` and whose subsequent elements are the original arguments. Then result of the call to `handleMethodInvocationError` is pushed onto the operand stack of the invoker.

Otherwise, if the selected method is not accessible (§5.4.4) to the current class, *invokedynamic* invokes the method named `handleMethodInvocationError` with descriptor `([Ljava/lang/Object;)Ljava/lang/Object;` on `objectref`, with an argument that is an object array whose zeroth element is the name of the referenced method, whose first element is the descriptor of the referenced method, whose second element is an instance of the class `IllegalAccessException` and whose subsequent elements are the
original arguments. Then result of the call to handleMethodInvocationError is pushed onto the operand stack of the invoker.

Otherwise, if the selected method is protected (§4.6), and it is a member of a superclass of the current class, and the method is not declared in the same run-time package (§5.3) as the current class, and the class of objectref is not the current class or a subclass of the current class, then invokedynamic invokes the method named handleMethodInvocationError with descriptor ([Ljava/lang/Object;)Ljava/lang/Object; on objectref, with an argument that is an object array whose zeroth element is the name of the referenced method, whose first element is the descriptor of the referenced method, whose second element is an instance of the class IllegalAccessException and whose subsequent elements are the original arguments. Then result of the call to handleMethodInvocationError is pushed onto the operand stack of the invoker.

Otherwise, each actual argument is cast to the corresponding argument type given in the descriptor of the resolved method. If any such cast fails, invokedynamic invokes the method named handleMethodInvocationError with descriptor ([Ljava/lang/Object;)Ljava/lang/Object; on objectref, with an argument that is an object array whose zeroth element is the name of the referenced method, whose first element is the descriptor of the resolved method, whose second element is an instance of the class ClassCastException and whose subsequent elements are the original argument. Then result of the call to handleMethodInvocationError is pushed onto the operand stack of the invoker.

Otherwise, if the method is synchronized, the monitor associated with objectref is acquired or reentered.

If the method is not native, the nargs argument values and objectref are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The objectref and the argument values are consecutively made the values of local variables of the new frame, with objectref in local variable 0, arg1 in local variable 1, and so on. The new frame is then made current, and the Java virtual machine pc is set to the opcode
of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done.

If the code that implements the method cannot be bound, `invokedynamic` invokes the method named `handleMethodInvocationError` with descriptor `([Ljava/lang/Object;)Ljava/lang/Object;` on `objectref`, with an argument that is an object array—whose zeroth element is the name of the referenced method, whose first element is the descriptor of the resolved method, whose second element is an instance of the class `UnsatisfiedLinkError` and whose subsequent elements are the original arguments. Then result of the call to `handleMethodInvocationError` is pushed onto the operand stack of the invoker.

Otherwise, the `nargs` argument values and `objectref` are popped from the operand stack and are passed as parameters to the code that implements the method. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the native method is synchronized, the monitor associated with `objectref` is released or exited as if by execution of a `monitorexit` instruction.

- If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

**Runtime Exceptions**

Otherwise, if `objectref` is `null`, the `invokedynamic` instruction throws a `NullPointerException`.

If the call is delegated to `handleMethodInvocationError` and that call results in an exception, then the `invokedynamic` instruction throws the exception that was passed as the second element of the array passed as an actual argument to `handleMethodInvocationError`. 
Notes
The `nargs` argument values and `objectref` are one-to-one with the first `nargs + 1` local variables.
**invokeinterface**

**Operation**  Invoke interface method

**Format**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>invokeinterface</td>
<td>indexbyte1</td>
<td>indexbyte2</td>
</tr>
<tr>
<td>count</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**

`invokeinterface = 185 (0xb9)`

**Operand**  `..., objectref, [arg1, [arg2 ...]]` ⇒

**Stack**  `...`

**Description**

The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to an interface method (§5.1), which gives the name and descriptor (§4.3.3) of the interface method as well as a symbolic reference to the interface in which the interface method is to be found. The named interface method is resolved (§5.4.3.4). The interface method must not be an instance initialization method (§3.9) or the class or interface initialization method (§3.9).

The `count` operand is an unsigned byte that must not be zero. The `objectref` must be of type `reference` and must be followed on the operand stack by `nargs` argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved interface method. The value of the fourth operand byte must always be zero.

Let `C` be the class of `objectref`. The actual method to be invoked is selected by the following lookup procedure:
invokeinterface (cont.)

- If \( C \) contains a declaration for an instance method with the same name and descriptor as the resolved method, then this is the method to be invoked, and the lookup procedure terminates.

- Otherwise, if \( C \) has a superclass, this same lookup procedure is performed recursively using the direct superclass of \( C \); the method to be invoked is the result of the recursive invocation of this lookup procedure.

- Otherwise, an AbstractMethodError is raised.

If the method is synchronized, the monitor associated with \( \text{object-ref} \) is acquired or reentered.

If the method is not native, the \( n\text{args} \) argument values and \( \text{object-ref} \) are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The \( \text{object-ref} \) and the argument values are consecutively made the values of local variables of the new frame, with \( \text{object-ref} \) in local variable 0, \( \text{arg1} \) in local variable 1 (or, if \( \text{arg1} \) is of type \( \text{long} \) or \( \text{double} \), in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine \( \text{pc} \) is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The \( n\text{args} \) argument values and \( \text{object-ref} \) are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns:
**invokeinterface (cont.)**

- If the native method is synchronized, the monitor associated with `objectref` is released or exited as if by execution of a `monitor-exit` instruction.

- If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

**Linking Exceptions**

During resolution of the symbolic reference to the interface method, any of the exceptions documented in §5.4.3.4 can be thrown.

**Runtime Exceptions**

Otherwise, if `objectref` is `null`, the `invokeinterface` instruction throws a `NullPointerException`.

Otherwise, if the class of `objectref` does not implement the resolved interface, `invokeinterface` throws an `IncompatibleClassChangeError`.

Otherwise, if no method matching the resolved name and descriptor is selected, `invokeinterface` throws an `AbstractMethodError`.

Otherwise, if the selected method is not public, `invokeinterface` throws an `IllegalAccessError`.

Otherwise, if the selected method is abstract, `invokeinterface` throws an `AbstractMethodError`.

Otherwise, if the selected method is native and the code that implements the method cannot be bound, `invokeinterface` throws an `UnsatisfiedLinkError`. 
**invokeinterface (cont.)**

**invokeinterface (cont.)**

**Notes**

The *count* operand of the *invokeinterface* instruction records a measure of the number of argument values, where an argument value of type *long* or type *double* contributes two units to the *count* value and an argument of any other type contributes one unit. This information can also be derived from the descriptor of the selected method. The redundancy is historical.

The fourth operand byte exists to reserve space for an additional operand used in certain of Sun’s implementations, which replace the *invokeinterface* instruction by a specialized pseudo-instruction at run time. It must be retained for backwards compatibility.

The *nargs* argument values and *objectref* are not one-to-one with the first *nargs* + 1 local variables. Argument values of types *long* and *double* must be stored in two consecutive local variables, thus more than *nargs* local variables may be required to pass *nargs* argument values to the invoked method.
**invokespecial**

**Operation**
Invoke instance method; special handling for superclass, private, and instance initialization method invocations

**Format**

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
<td></td>
</tr>
<tr>
<td>indexbyte2</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**

`invokespecial = 183 (0xb7)`

**Operand**

`... objectref, [arg1, [arg2 ...]] ⇒`

**Stack**

`...`

**Description**

The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). Finally, if the resolved method is protected (§4.6), and it is a member of a superclass of the current class, and the method is not declared in the same run-time package (§5.3) as the current class, then the class of `objectref` must be either the current class or a subclass of the current class.

Next, the resolved method is selected for invocation unless all of the following conditions are true:

- The ACC_SUPER flag (see Table 4.1, “Class access and property modifiers”) is set for the current class.
- The class of the resolved method is a superclass of the current class.
• The resolved method is not an instance initialization method
  (§3.9).

If the above conditions are true, the actual method to be
invoked is selected by the following lookup procedure. Let \( C \) be the
direct superclass of the current class:

If \( C \) contains a declaration for an instance method with the
same name and descriptor as the resolved method, then this method
will be invoked. The lookup procedure terminates.

Otherwise, if \( C \) has a superclass, this same lookup procedure is
performed recursively using the direct superclass of \( C \). The method
to be invoked is the result of the recursive invocation of this lookup
procedure.

Otherwise, an AbstractMethodError is raised.

The \( \text{objectref} \) must be of type \( \text{reference} \) and must be followed on
the operand stack by \( \text{nargs} \) argument values, where the number,
type, and order of the values must be consistent with the descriptor
of the selected instance method.

If the method is synchronized, the monitor associated with
\( \text{objectref} \) is acquired or reentered.

If the method is not native, the \( \text{nargs} \) argument values and \( \text{objectref} \)
are popped from the operand stack. A new frame is created on
the Java virtual machine stack for the method being invoked. The
\( \text{objectref} \) and the argument values are consecutively made the val-
ues of local variables of the new frame, with \( \text{objectref} \) in local vari-
able 0, \( \text{arg1} \) in local variable 1 (or, if \( \text{arg1} \) is of type \( \text{long} \) or
\( \text{double} \), in local variables 1 and 2), and so on. Any argument value
that is of a floating-point type undergoes value set conversion
(§3.8.3) prior to being stored in a local variable. The new frame is
then made current, and the Java virtual machine \( \text{pc} \) is set to the
opcode of the first instruction of the method to be invoked. Execu-
tion continues with the first instruction of the method.
invokespecial (cont.)

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The nargs argument values and objectref are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

• If the native method is synchronized, the monitor associated with objectref is released or exited as if by execution of a monitor-exit instruction.

• If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in Section 5.4.3.3 can be thrown.

Otherwise, if the resolved method is an instance initialization method, and the class in which it is declared is not the class symbolically referenced by the instruction, a NoSuchMethodError is thrown.

Otherwise, if the resolved method is a class (static) method, the invokespecial instruction throws an IncompatibleClassChangeError.

Otherwise, if no method matching the resolved name and descriptor is selected, invokespecial throws an AbstractMethodError.

Otherwise, if the selected method is abstract, invokespecial throws an AbstractMethodError.
invokespecial (cont.)

Runtime Exceptions

Otherwise, if objectref is null, the invokespecial instruction throws a NullPointerException.

Otherwise, if the selected method is native and the code that implements the method cannot be bound, invokespecial throws an UnsatisfiedLinkError.

Notes

The difference between the invokespecial and the invokevirtual instructions is that invokevirtual invokes a method based on the class of the object. The invokespecial instruction is used to invoke instance initialization methods (§3.9) as well as private methods and methods of a superclass of the current class.

The invokespecial instruction was named invokenonvirtual prior to Sun’s JDK release 1.0.2.

The nargs argument values and objectref are not one-to-one with the first nargs + 1 local variables. Argument values of types long and double must be stored in two consecutive local variables, thus more than nargs local variables may be required to pass nargs argument values to the invoked method.
**invokestatic**  

**Operation** Invoke a class (static) method

**Format**

| invokestatic |
| indexbyte1 |
| indexbyte2 |

**Forms** $invokestatic = 184 \ (0xb8)$

**Operand Stack** ...

**Description**
The unsigned $indexbyte1$ and $indexbyte2$ are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is $(indexbyte1 \ll 8) \mid indexbyte2$. The runtime constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The method must not be the class or interface initialization method (§3.9). It must be static, and therefore cannot be abstract.

On successful resolution of the method, the class that declared the resolved method is initialized (§5.5) if that class has not already been initialized.

The operand stack must contain $nargs$ argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved method.

If the method is synchronized, the monitor associated with the resolved class is acquired or reentered.
invokestatic (cont.)

If the method is not native, the nargs argument values are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The nargs argument values are consecutively made the values of local variables of the new frame, with arg1 in local variable 0 (or, if arg1 is of type long or double, in local variables 0 and 1) and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The nargs argument values are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the native method is synchronized, the monitor associated with the resolved class is released or exited as if by execution of a monitorexit instruction.

- If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in Section 5.4.3.3 can be thrown.
Otherwise, if the resolved method is an instance method, the invokestatic instruction throws an IncompatibleClassChangeError.

**Runtime Exceptions**

Otherwise, if execution of this invokestatic instruction causes initialization of the referenced class, invokestatic may throw an Error as detailed in Section 2.17.5.

Otherwise, if the resolved method is native and the code that implements the method cannot be bound, invokestatic throws an UnsatisfiedLinkError.

**Notes**

The nargs argument values are not one-to-one with the first nargs local variables. Argument values of types long and double must be stored in two consecutive local variables, thus more than nargs local variables may be required to pass nargs argument values to the invoked method.
invokevirtual

**Operation**  Invoke instance method; dispatch based on class

**Format**

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

**Forms**

invokevirtual = 182 (0xb6)

**Operand**

..., objectref, [arg1, [arg2 ...]] ⇒

**Stack**

...

**Description**

The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The method must not be an instance initialization method (§3.9) or the class or interface initialization method (§3.9). Finally, if the resolved method is protected (§4.6), and it is a member of a superclass of the current class, and the method is not declared in the same run-time package (§5.3) as the current class, then the class of `objectref` must be either the current class or a subclass of the current class.

Let C be the class of `objectref`. The actual method to be invoked is selected by the following lookup procedure:

- If C contains a declaration for an instance method M with the same name and descriptor as the resolved method, and M overrides the resolved method, then M is the method to be invoked, and the lookup procedure terminates.
invokevirtual (cont.)

- Otherwise, if \( C \) has a superclass, this same lookup procedure is performed recursively using the direct superclass of \( C \); the method to be invoked is the result of the recursive invocation of this lookup procedure.

- Otherwise, an AbstractMethodError is raised.

The objectref must be followed on the operand stack by nargs argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is synchronized, the monitor associated with objectref is acquired or reentered.

If the method is not native, the nargs argument values and objectref are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The objectref and the argument values are consecutively made the values of local variables of the new frame, with objectref in local variable 0, arg1 in local variable 1 (or, if arg1 is of type long or double, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The nargs argument values and objectref are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:
**invokevirtual (cont.)**

- If the native method is synchronized, the monitor associated with `objectref` is released or exited as if by execution of a `monitor-exit` instruction.

- If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

**Linking Exceptions**

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in Section 5.4.3.3 can be thrown.

Otherwise, if the resolved method is a class (static) method, the `invokevirtual` instruction throws an `IncompatibleClassChangeError`.

**Runtime Exceptions**

Otherwise, if `objectref` is `null`, the `invokevirtual` instruction throws a `NullPointerException`.

Otherwise, if no method matching the resolved name and descriptor is selected, `invokevirtual` throws an `AbstractMethodError`. Otherwise, if the selected method is abstract, `invokevirtual` throws an `AbstractMethodError`.

Otherwise, if the selected method is native and the code that implements the method cannot be bound, `invokevirtual` throws an `UnsatisfiedLinkError`.

**Notes**

The `nargs` argument values and `objectref` are not one-to-one with the first `nargs + 1` local variables. Argument values of types `long` and `double` must be stored in two consecutive local variables, thus more than `nargs` local variables may be required to pass `nargs` argument values to the invoked method.
**Operation**  Boolean OR int

**Format**  

<table>
<thead>
<tr>
<th>\textit{ior}</th>
</tr>
</thead>
</table>

**Forms**  \textit{ior} = 128 (0x80)

**Operand**  \textit{..., \textit{value1}, \textit{value2} ⇒}

**Stack**  \textit{..., \textit{result}}

**Description**  Both \textit{value1} and \textit{value2} must be of type int. They are popped from the operand stack. An int \textit{result} is calculated by taking the bitwise inclusive OR of \textit{value1} and \textit{value2}. The \textit{result} is pushed onto the operand stack.
irem  

Operation  
Remainder int

Format  

Forms  
irem = 112 (0x70)

Operand  
..., value1, value2 ⇒

Stack  
..., result

Description  
Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 − (value1 / value2) * value2. The result is pushed onto the operand stack.

The result of the irem instruction is such that (a/b)*b + (a%b) is equal to a. This identity holds even in the special case in which the dividend is the negative int of largest possible magnitude for its type and the divisor is −1 (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

Runtime Exception  
If the value of the divisor for an int remainder operator is 0, irem throws an ArithmeticException.
**Operation** Return int from method

**Format**

```plaintext
ireturn
```

**Forms**

```
ireturn = 172 (0xac)
```

**Operand Stack**

```
..., value ⇒
```

**Description**

The current method must have return type boolean, byte, short, char, or int. The `value` must be of type int. If the current method is a synchronized method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a `monitorexit` instruction. If no exception is thrown, `value` is popped from the operand stack of the current frame (§3.6) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

**Runtime Exceptions**

If the current method is a synchronized method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, `ireturn` throws an `IllegalMonitorStateException`. This can happen, for example, if a synchronized method contains a `monitorexit` instruction, but no `monitorenter` instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in Section 8.13 and if the first of those rules is violated during invocation of the current method, then `ireturn` throws an `IllegalMonitorStateException`.
### ishl

**Operation**  
Shift left int

**Format**  

```
  ishl
```

**Forms**  

ishl = 120 (0x78)

**Operand**  

..., value1, value2 ⇒

**Stack**  

..., result

**Description**  
Both value1 and value2 must be of type int. The values are popped from the operand stack. An int result is calculated by shifting value1 left by s bit positions, where s is the value of the low 5 bits of value2. The result is pushed onto the operand stack.

**Notes**  
This is equivalent (even if overflow occurs) to multiplication by 2 to the power s. The shift distance actually used is always in the range 0 to 31, inclusive, as if value2 were subjected to a bitwise logical AND with the mask value 0x1f.
**ishr**

**Operation**  
Arithmetic shift right \( \text{int} \)

**Format**  

```
| ishr |
```

**Forms**  
\( ishr = 122 \ (0x7a) \)

**Operand**  
\( \ldots, \text{value1, value2} \Rightarrow \)

**Stack**  
\( \ldots, \text{result} \)

**Description**  
Both \( \text{value1} \) and \( \text{value2} \) must be of type \( \text{int} \). The values are popped from the operand stack. An \( \text{int} \) \( \text{result} \) is calculated by shifting \( \text{value1} \) right by \( s \) bit positions, with sign extension, where \( s \) is the value of the low 5 bits of \( \text{value2} \). The \( \text{result} \) is pushed onto the operand stack.

**Notes**  
The resulting value is \( \left\lfloor \frac{\text{value1}}{2^s} \right\rfloor \), where \( s \) is \( \text{value2} \) \& 0x1f. For nonnegative \( \text{value1} \), this is equivalent to truncating \( \text{int} \) division by 2 to the power \( s \). The shift distance actually used is always in the range 0 to 31, inclusive, as if \( \text{value2} \) were subjected to a bitwise logical AND with the mask value 0x1f.
**istore**

**Operation** Store int into local variable

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>istore</td>
<td>index</td>
</tr>
</tbody>
</table>

**Forms**

`istore = 54 (0x36)`

**Operand**

`..., value ⇒`

**Stack**

`...`

**Description** The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The value on the top of the operand stack must be of type int. It is popped from the operand stack, and the value of the local variable at index is set to value.

**Notes** The istore opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
**istore_<n>**

**Operation**  
Store int into local variable

**Format**  

<table>
<thead>
<tr>
<th>Format</th>
<th>istore_&lt;n&gt;</th>
</tr>
</thead>
</table>

**Forms**  
`istore_0 = 59 (0x3b)`  
`istore_1 = 60 (0x3c)`  
`istore_2 = 61 (0x3d)`  
`istore_3 = 62 (0x3e)`

**Operand**  
...

**Stack**  
...

**Description**  
The `<n>` must be an index into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type `int`. It is popped from the operand stack, and the value of the local variable at `<n>` is set to `value`.

**Notes**  
Each of the `istore_<n>` instructions is the same as `istore` with an index of `<n>`, except that the operand `<n>` is implicit.
**Operation**  Subtract int

**Format**  

```
isub
```

**Forms**  

\( isub = 100 \ (0x64) \)

**Operand**  

\( \ldots,\ value1,\ value2 \Rightarrow \)

**Stack**  

\( \ldots,\ result \)

**Description**  

Both \( value1 \) and \( value2 \) must be of type int. The values are popped from the operand stack. The int result is \( value1 - value2 \). The result is pushed onto the operand stack.

For int subtraction, \( a - b \) produces the same result as \( a + (-b) \). For int values, subtraction from zero is the same as negation.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type int. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical difference of the two values.

Despite the fact that overflow may occur, execution of an \( isub \) instruction never throws a runtime exception.
**Operation** Logical shift right int

**Format**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>iushr</td>
</tr>
</tbody>
</table>

**Forms**

\[ iushr = 124 \ (0x7c) \]

**Operand** \( \ldots, \text{value1, value2} \Rightarrow \)

**Stack** \( \ldots, \text{result} \)

**Description** Both value1 and value2 must be of type int. The values are popped from the operand stack. An int result is calculated by shifting value1 right by \( s \) bit positions, with zero extension, where \( s \) is the value of the low 5 bits of value2. The result is pushed onto the operand stack.

**Notes** If value1 is positive and \( s = \text{value2} \& 0x1f \), the result is the same as that of value1 \( \gg s \); if value1 is negative, the result is equal to the value of the expression \( (\text{value1} \gg s) + (2 << \sim s) \). The addition of the \( 2 << \sim s \) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive.
**ixor**

**Operation** Boolean XOR int

**Format**

| ixor |

**Forms** ixor = 130 (0x82)

**Operand** ...., value1, value2 ⇒

**Stack** ...., result

**Description** Both value1 and value2 must be of type int. They are popped from the operand stack. An int result is calculated by taking the bitwise exclusive OR of value1 and value2. The result is pushed onto the operand stack.
**jsr**

**Operation**  
Jump subroutine

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jsr</td>
<td></td>
</tr>
<tr>
<td>branchbyte1</td>
<td></td>
</tr>
<tr>
<td>branchbyte2</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**

$jsr = 168 (0xa8)$

**Operand** … ⇒

**Stack** …, address

**Description**  
The *address* of the opcode of the instruction immediately following this *jsr* instruction is pushed onto the operand stack as a value of type *returnAddress*. The unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is $(branchbyte1 \ll 8) | branchbyte2$. Execution proceeds at that offset from the address of this *jsr* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr* instruction.

**Notes**  
The *jsr* instruction is used with the *ret* instruction in the implementation of the *finally* clauses of the Java programming language (see Section 7.13, “Compiling *finally*”). Note that *jsr* pushes the address onto the operand stack and *ret* gets it out of a local variable. This asymmetry is intentional.
**jsr_w**

**Operation** Jump subroutine (wide index)

**Format**

<table>
<thead>
<tr>
<th>jsr_w</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
<tr>
<td>branchbyte3</td>
</tr>
<tr>
<td>branchbyte4</td>
</tr>
</tbody>
</table>

**Forms**

\[ jsr_w = 201 \ (0xc9) \]

**Operand**

… ⇒

**Stack**

…, \textit{address}  

**Description**

The \textit{address} of the opcode of the instruction immediately following this \textit{jsr_w} instruction is pushed onto the operand stack as a value of type \texttt{returnAddress}. The unsigned \textit{branchbyte1}, \textit{branchbyte2}, \textit{branchbyte3}, and \textit{branchbyte4} are used to construct a signed 32-bit offset, where the offset is \((\text{branchbyte1} \ll 24) | (\text{branchbyte2} \ll 16) | (\text{branchbyte3} \ll 8) | \text{branchbyte4}\). Execution proceeds at that offset from the address of this \textit{jsr_w} instruction. The target address must be that of an opcode of an instruction within the method that contains this \textit{jsr_w} instruction.

**Notes**

The \textit{jsr_w} instruction is used with the \textit{ret} instruction in the implementation of the \texttt{finally} clauses of the Java programming language (see Section 7.13, “Compiling \texttt{finally}”). Note that \textit{jsr_w} pushes the address onto the operand stack and \textit{ret} gets it out of a local variable. This asymmetry is intentional.

Although the \textit{jsr_w} instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes (§4.10). This limit may be raised in a future release of the Java virtual machine.
**l2d**

**Operation**  Convert `long` to `double`

**Format**

```
  l2d
```

**Forms**  

`l2d = 138 (0x8a)`

**Operand**  

`...`  

**Stack**  

`...`  

**Description**  

The `value` on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to a `double` `result` using IEEE 754 round to nearest mode. The `result` is pushed onto the operand stack.

**Notes**  

The `l2d` instruction performs a widening primitive conversion (§2.6.2) that may lose precision because values of type `double` have only 53 significand bits.
<table>
<thead>
<tr>
<th><strong>Operation</strong></th>
<th>Convert <code>long</code> to <code>float</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format</strong></td>
<td><code>l2f</code></td>
</tr>
<tr>
<td><strong>Forms</strong></td>
<td><code>l2f = 137 (0x89)</code></td>
</tr>
<tr>
<td><strong>Operand</strong></td>
<td>..., <code>value</code> ⇒</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>..., <code>result</code></td>
</tr>
</tbody>
</table>

**Description**
The `value` on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to a `float` `result` using IEEE 754 round to nearest mode. The `result` is pushed onto the operand stack.

**Notes**
The `l2f` instruction performs a widening primitive conversion (§2.6.2) that may lose precision because values of type `float` have only 24 significand bits.
l2i

Operation  Convert long to int

Format  l2i

Forms  l2i = 136 (0x88)

Operand  ..., value ⇒
Stack  ..., result

Description  The value on the top of the operand stack must be of type long. It is popped from the operand stack and converted to an int result by taking the low-order 32 bits of the long value and discarding the high-order 32 bits. The result is pushed onto the operand stack.

Notes  The l2i instruction performs a narrowing primitive conversion (§2.6.3). It may lose information about the overall magnitude of value. The result may also not have the same sign as value.
**ladd**

**Operation**  Add long

**Format**  

```
  ladd
```

**Forms**  

\[ ladd = 97 \ (0\times61) \]

**Operand**  

\[ \ldots, \text{value1, value2} \Rightarrow \]

**Stack**  

\[ \ldots, \text{result} \]

**Description**  
Both \textit{value1} and \textit{value2} must be of type \texttt{long}. The values are popped from the operand stack. The \texttt{long} \textit{result} is \textit{value1} + \textit{value2}. The \textit{result} is pushed onto the operand stack.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type \texttt{long}. If overflow occurs, the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an \textit{ladd} instruction never throws a runtime exception.
**laload**

**Operation**  Load long from array

**Format**  

```
lload
```

**Forms**  

```
lload = 47 (0x2f)
```

**Operand Stack**  

```
..., arrayref, index ⇒
```

**Description**  
The `arrayref` must be of type reference and must refer to an array whose components are of type `long`. The `index` must be of type `int`. Both `arrayref` and `index` are popped from the operand stack. The `long` value in the component of the array at `index` is retrieved and pushed onto the operand stack.

**Runtime Exceptions**  

If `arrayref` is null, `lload` throws a NullPointerException.

Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `lload` instruction throws an ArrayIndexOutOfBoundsException.
land

**Operation**  Boolean AND long

**Format**  

```
land
```

**Forms**  

land = 127 (0x7f)

**Operand**  

..., value1, value2 ➞

**Stack**  

..., result

**Description**  Both value1 and value2 must be of type long. They are popped from the operand stack. A long result is calculated by taking the bitwise AND of value1 and value2. The result is pushed onto the operand stack.
lastore

Operation  Store into long array

Format  

Forms  lastore = 80 (0x50)

Operand  ..., arrayref, index, value ⇒

Stack  ...

Description  The arrayref must be of type reference and must refer to an array whose components are of type long. The index must be of type int, and value must be of type long. The arrayref, index, and value are popped from the operand stack. The long value is stored as the component of the array indexed by index.

Runtime Exceptions  

If arrayref is null, lastore throws a NullPointerException.

Otherwise, if index is not within the bounds of the array referenced by arrayref, the lastore instruction throws an ArrayIndexOutOfBoundsException.
**lcmp**

**Operation**  
Compare long

**Format**  
```
lcmp
```

**Forms**  
lcmp = 148 (0x94)

**Operand**  
```
..., value1, value2 ⇒
```

**Stack**  
```
..., result
```

**Description**  
Both `value1` and `value2` must be of type `long`. They are both popped from the operand stack, and a signed integer comparison is performed. If `value1` is greater than `value2`, the `int` value 1 is pushed onto the operand stack. If `value1` is equal to `value2`, the `int` value 0 is pushed onto the operand stack. If `value1` is less than `value2`, the `int` value –1 is pushed onto the operand stack.
### lconst_<l>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Push long constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>lconst_&lt;l&gt;</td>
</tr>
<tr>
<td>Forms</td>
<td>lconst_0 = 9 (0x9)</td>
</tr>
<tr>
<td></td>
<td>lconst_1 = 10 (0xa)</td>
</tr>
<tr>
<td>Operand</td>
<td>... ⇒</td>
</tr>
<tr>
<td>Stack</td>
<td>..., &lt;l&gt;</td>
</tr>
<tr>
<td>Description</td>
<td>Push the long constant &lt;l&gt; (0 or 1) onto the operand stack.</td>
</tr>
</tbody>
</table>
**ldc**

**Operation**  
Push item from runtime constant pool

**Format**  

<table>
<thead>
<tr>
<th>ldc</th>
<th>index</th>
</tr>
</thead>
</table>

**Forms**  
\( ldc = 18 \ (0x12) \)

**Operand**  
… ⇒

**Stack**  
…, value

**Description**  
The \textit{index} is an unsigned byte that must be a valid index into the runtime constant pool of the current class (§3.6). The runtime constant pool entry at \textit{index} either must be a runtime constant of type \texttt{int} or \texttt{float}, or must be a symbolic reference to a class (§5.4.3.1) or a string literal (§5.1).

If the runtime constant pool entry is a runtime constant of type \texttt{int} or \texttt{float}, the numeric \textit{value} of that runtime constant is pushed onto the operand stack as an \texttt{int} or \texttt{float}, respectively.

Otherwise, if the runtime constant pool entry is a reference to an instance of class \texttt{String} representing a string literal (§5.1), then a reference to that instance, \textit{value}, is pushed onto the operand stack.

Otherwise, the runtime constant pool entry must be a symbolic reference to a class (§4.4.1). The named class is resolved (§5.4.3.1) and a reference to the Class object representing that class, \textit{value}, is pushed onto the operand stack.

**Linking Exceptions**  
During resolution of the symbolic reference to the class, any of the exceptions pertaining to class resolution documented in Section 5.4.3.1 can be thrown.
Notes  The `ldc` instruction can only be used to push a value of type `float` taken from the float value set (§3.3.2) because a constant of type `float` in the constant pool (§4.4.4) must be taken from the float value set.
**ldc_w**

**Operation** Push item from runtime constant pool (wide index)

**Format**

<table>
<thead>
<tr>
<th>Operation</th>
<th>ldc_w</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexbyte1</td>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms**

$ldc_w = 19 \ (0x13)$

**Operand Stack**

... ⇒ ...., value

**Description**

The unsigned $indexbyte1$ and $indexbyte2$ are assembled into an unsigned 16-bit index into the runtime constant pool of the current class (§3.6), where the value of the index is calculated as $(indexbyte1 \ll 8) \mid indexbyte2$. The index must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at the index either must be a runtime constant of type `int` or `float`, or must be a symbolic reference to a class (§5.4.3.1) or a string literal (§5.1).

If the runtime constant pool entry is a runtime constant of type `int` or `float`, the numeric value of that runtime constant is pushed onto the operand stack as an `int` or `float`, respectively.

Otherwise, if the runtime constant pool entry is a reference to an instance of class `String` representing a string literal (§5.1), then a reference to that instance, `value`, is pushed onto the operand stack.

Otherwise, the runtime constant pool entry must be a symbolic reference to a class (§4.4.1). The named class is resolved (§5.4.3.1) and a reference to the `Class` object representing that class, `value`, is pushed onto the operand stack.
**Linking Exceptions**

During resolution of the symbolic reference to the class, any of the exceptions pertaining to class resolution documented in Section 5.4.3.1 can be thrown.

**Notes**

The `ldc_w` instruction is identical to the `ldc` instruction except for its wider runtime constant pool index.

The `ldc_w` instruction can only be used to push a value of type `float` taken from the float value set (§3.3.2) because a constant of type `float` in the constant pool (§4.4.4) must be taken from the float value set.
\textit{ldc2\_w} \hspace{1cm} \textit{ldc2\_w}

\textbf{Operation} \hspace{0.5cm} \text{Push long or double from runtime constant pool (wide index)}

\textbf{Format} \\
\begin{tabular}{|c|}
\hline
\textit{ldc2\_w} \\
\hline
indexbyte1 \\
\hline
indexbyte2 \\
\hline
\end{tabular}

\textbf{Forms} \hspace{0.5cm} \textit{ldc2\_w} = 20 (0x14)

\textbf{Operand} \hspace{1cm} \ldots \Rightarrow \\
\textbf{Stack} \hspace{1cm} \ldots, \textit{value}

\textbf{Description} \hspace{0.5cm} \text{The unsigned} \indexbyte1 \text{ and} \indexbyte2 \text{ are assembled into an unsigned 16-bit index into the runtime constant pool of the current class (§3.6), where the value of the index is calculated as} \text{(indexbyte1} \ll 8 \text{) | indexbyte2}. \text{The index must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at the index must be a runtime constant of type long or double (§5.1). The numeric value of that runtime constant is pushed onto the operand stack as a long or double, respectively.}

\textbf{Notes} \hspace{0.5cm} \text{Only a wide-index version of the \textit{ldc2\_w} instruction exists; there is no \textit{ldc2} instruction that pushes a long or double with a single-byte index. The \textit{ldc2\_w} instruction can only be used to push a value of type double taken from the double value set (§3.3.2) because a constant of type double in the constant pool (§4.4.5) must be taken from the double value set.}
ldiv

Operation  Divide long

Format  

Forms  \( ldiv = 109 \ (0x6d) \)

Operand  \( \ldots, value1, value2 \Rightarrow \)

Stack  \( \ldots, result \)

Description  Both \( value1 \) and \( value2 \) must be of type \( long \). The values are popped from the operand stack. The \( long \) \( result \) is the value of the Java programming language expression \( value1 / value2 \). The \( result \) is pushed onto the operand stack.

A \( long \) division rounds towards 0; that is, the quotient produced for \( long \) values in \( n / d \) is a \( long \) value \( q \) whose magnitude is as large as possible while satisfying \( |d \cdot q| \leq |n| \). Moreover, \( q \) is positive when \( |n| \geq |d| \) and \( n \) and \( d \) have the same sign, but \( q \) is negative when \( |n| \geq |d| \) and \( n \) and \( d \) have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for the \( long \) type and the divisor is \(-1\), then overflow occurs and the result is equal to the dividend; despite the overflow, no exception is thrown in this case.

Runtime Exception  If the value of the divisor in a \( long \) division is 0, \( ldiv \) throws an \( ArithmeticException \).
**lload**

**Operation**  Load `long` from local variable

**Format**

```
<table>
<thead>
<tr>
<th>lload</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
</tr>
</tbody>
</table>
```

**Forms**  
`lload = 22 (0x16)`

**Operand Stack**  
`... ⇒ ...`, `value`

**Description**  The `index` is an unsigned byte. Both `index` and `index + 1` must be indices into the local variable array of the current frame (§3.6). The local variable at `index` must contain a `long`. The `value` of the local variable at `index` is pushed onto the operand stack.

**Notes**  The `lload` opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
\textbf{\textit{lload}_{<n>}} \hspace{1cm} \textbf{\textit{lload}_{<n>}}

\textbf{Operation} \hspace{1cm} \text{Load } \textbf{long} \text{ from local variable}

\textbf{Format} \hspace{1cm} \text{\textit{lload}_{<n>}}

\textbf{Forms} \hspace{1cm} \textit{\textit{lload} \_0} = 30 \ (0x1e) \\
\textit{\textit{lload} \_1} = 31 \ (0x1f) \\
\textit{\textit{lload} \_2} = 32 \ (0x20) \\
\textit{\textit{lload} \_3} = 33 \ (0x21)

\textbf{Operand} \hspace{1cm} \text{… ⇒}
\textbf{Stack} \hspace{1cm} \text{…, \textit{value}}

\textbf{Description} \hspace{1cm} \text{Both } <n> \text{ and } <n> + 1 \text{ must be indices into the local variable array of the current frame (§3.6). The local variable at } <n> \text{ must contain a } \textbf{long}. \text{ The } \textit{value} \text{ of the local variable at } <n> \text{ is pushed onto the operand stack.}

\textbf{Notes} \hspace{1cm} \text{Each of the } \textit{lload}_{<n>} \text{ instructions is the same as } \textit{lload} \text{ with an index of } <n>, \text{ except that the operand } <n> \text{ is implicit.}
### lmul

**Operation**  Multiply `long`

**Format**  

**Forms**  

`lmul = 105 (0x69)`

**Operand**  

`..., value1, value2 =>`

**Stack**  

`..., result`

**Description**  

Both `value1` and `value2` must be of type `long`. The values are popped from the operand stack. The `long` result is `value1 * value2`. The `result` is pushed onto the operand stack.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type `long`. If overflow occurs, the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an `lmul` instruction never throws a runtime exception.
**lneg**

**Operation**  
Negate `long`

**Format**  
`lneg`

**Forms**  
`lneg = 117 (0x75)`

**Operand**  
`..., value ⇒`

**Stack**  
`..., result`

**Description**  
The `value` must be of type `long`. It is popped from the operand stack. The `long` `result` is the arithmetic negation of `value`, \(-value\). The `result` is pushed onto the operand stack.

For `long` values, negation is the same as subtraction from zero. Because the Java virtual machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative `long` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all `long` values `x`, \(-x\) equals \((\sim x) + 1\).
lookupswitch

**Operation**  Access jump table by key match and jump

**Format**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lookupswitch</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;0-3 byte pad&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>defaultbyte1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>defaultbyte2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>defaultbyte3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>defaultbyte4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>npairs1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>npairs2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>npairs3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>npairs4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>match-offset pairs…</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Forms**  

`lookupswitch = 171 (0xab)`

**Operand**  

`..., key ⇒`

**Stack**  

`...`

**Description**  

A `lookupswitch` is a variable-length instruction. Immediately after the `lookupswitch` opcode, between zero and three null bytes (zeroed bytes, not the null object) are inserted as padding. The number of null bytes is chosen so that the `defaultbyte1` begins at an address that is a multiple of four bytes from the start of the current method (the opcode of its first instruction). Immediately after the padding follow a series of signed 32-bit values: `default`, `npairs`, and then `npairs` pairs of signed 32-bit values. The `npairs` must be greater than or equal to 0. Each of the `npairs` pairs consists of an integer `match` and a signed 32-bit `offset`. Each of these signed 32-bit values is constructed from four unsigned bytes as `(byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4.`
The table match-offset pairs of the `lookupswitch` instruction must be sorted in increasing numerical order by `match`.

The `key` must be of type `int` and is popped from the operand stack. The `key` is compared against the `match` values. If it is equal to one of them, then a target address is calculated by adding the corresponding `offset` to the address of the opcode of this `lookupswitch` instruction. If the `key` does not match any of the `match` values, the target address is calculated by adding `default` to the address of the opcode of this `lookupswitch` instruction. Execution then continues at the target address.

The target address that can be calculated from the offset of each `match-offset` pair, as well as the one calculated from `default`, must be the address of an opcode of an instruction within the method that contains this `lookupswitch` instruction.

**Notes**

The alignment required of the 4-byte operands of the `lookupswitch` instruction guarantees 4-byte alignment of those operands if and only if the method that contains the `lookupswitch` is positioned on a 4-byte boundary.

The `match-offset` pairs are sorted to support lookup routines that are quicker than linear search.
### lor

**Operation**  
Boolean OR long

**Format**  

| lor |

**Forms**  
lor = 129 (0x81)

**Operand**  
..., value1, value2 ⇒

**Stack**  
..., result

**Description**  
Both value1 and value2 must be of type long. They are popped from the operand stack. A long result is calculated by taking the bitwise inclusive OR of value1 and value2. The result is pushed onto the operand stack.
**lrem**

**Operation**  
Remainder `long`

**Format**  
```
lrem
```

**Forms**  
```
lrem = 113 (0x71)
```

**Operand Stack**  
```
..., value1, value2 ⇒
```

**Description**  
Both `value1` and `value2` must be of type `long`. The values are popped from the operand stack. The `long` result is `value1 − (value1 / value2) * value2`. The result is pushed onto the operand stack.

The result of the `lrem` instruction is such that `(a/b)*b + (a%b)` is equal to a. This identity holds even in the special case in which the dividend is the negative `long` of largest possible magnitude for its type and the divisor is −1 (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive; moreover, the magnitude of the result is always less than the magnitude of the divisor.

**Runtime Exception**  
If the value of the divisor for a `long` remainder operator is 0, `lrem` throws an `ArithmeticException`.
lreturn

Operation  Return long from method

Format  

Forms  \( lreturn = 173 \ (0xad) \)

Operand  \( \ldots, \text{value} \Rightarrow \)
Stack  [empty]

Description  The current method must have return type long. The value must be of type long. If the current method is a synchronized method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a monitorexit instruction. If no exception is thrown, value is popped from the operand stack of the current frame (§3.6) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

Runtime Exceptions  If the current method is a synchronized method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, lreturn throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monitorenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in Section 8.13 and if the first of those rules is violated during invocation of the current method, then lreturn throws an IllegalMonitorStateException.
The Java Virtual Machine Instruction Set

**lshl**

**Operation**  Shift left long

**Format**  

**Form**

**Operand**  ..., value1, value2 ⇒

**Stack**  ..., result

**Description**  The value1 must be of type long, and value2 must be of type int. The values are popped from the operand stack. A long result is calculated by shifting value1 left by s bit positions, where s is the low 6 bits of value2. The result is pushed onto the operand stack.

**Notes**  This is equivalent (even if overflow occurs) to multiplication by 2 to the power s. The shift distance actually used is therefore always in the range 0 to 63, inclusive, as if value2 were subjected to a bitwise logical AND with the mask value 0x3f.
**lshr**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arithmetic shift right long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td></td>
</tr>
<tr>
<td>Forms</td>
<td>lshr = 123 (0x7b)</td>
</tr>
<tr>
<td>Operand Stack</td>
<td>..., value1, value2 ⇒</td>
</tr>
<tr>
<td>Stack</td>
<td>..., result</td>
</tr>
</tbody>
</table>

**Description**
The `value1` must be of type `long`, and `value2` must be of type `int`. The values are popped from the operand stack. A `long` result is calculated by shifting `value1` right by `s` bit positions, with sign extension, where `s` is the value of the low 6 bits of `value2`. The result is pushed onto the operand stack.

**Notes**
The resulting value is \(\left\lfloor \frac{value1}{2^s} \right\rfloor\), where `s` is `value2` & 0x3f. For nonnegative `value1`, this is equivalent to truncating `long` division by 2 to the power `s`. The shift distance actually used is therefore always in the range 0 to 63, inclusive, as if `value2` were subjected to a bitwise logical AND with the mask value 0x3f.
**lstore**

**Operation**  
Store `long` into local variable

**Format**

| lstore | index |

**Forms**  
lstore = 55 (0x37)

**Operand**  
..., value ⇒

**Stack**  
...

**Description**  
The `index` is an unsigned byte. Both `index` and `index + 1` must be indices into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at `index` and `index + 1` are set to `value`.

**Notes**  
The `lstore` opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
### lstore_<n>

**Operation**  
Store `long` into local variable

**Format**

<table>
<thead>
<tr>
<th>lstore_&lt;n&gt;</th>
</tr>
</thead>
</table>

**Forms**

- `lstore_0 = 63 (0x3f)`
- `lstore_1 = 64 (0x40)`
- `lstore_2 = 65 (0x41)`
- `lstore_3 = 66 (0x42)`

**Operand Stack**

\[\ldots, \text{value} \Rightarrow \ldots\]

**Description**

Both `<n>` and `<n> + 1` must be indices into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at `<n>` and `<n> + 1` are set to `value`.

**Notes**

Each of the `lstore_<n>` instructions is the same as `lstore` with an *index* of `<n>`, except that the operand `<n>` is implicit.
The Java Virtual Machine Instruction Set

**lsub**

**Operation** Subtract long

**Format**

<table>
<thead>
<tr>
<th>lsub</th>
</tr>
</thead>
</table>

**Forms**

\[ lsub = 101 \text{ (0x65)} \]

**Operand** \( \ldots, \text{value1, value2} \Rightarrow \)

**Stack** \( \ldots, \text{result} \)

**Description**

Both \( \text{value1} \) and \( \text{value2} \) must be of type \( \text{long} \). The values are popped from the operand stack. The \( \text{long} \) result is \( \text{value1} - \text{value2} \). The result is pushed onto the operand stack.

For \( \text{long} \) subtraction, \( a-b \) produces the same result as \( a+(\neg b) \). For \( \text{long} \) values, subtraction from zero is the same as negation.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type \( \text{long} \). If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an \( lsub \) instruction never throws a runtime exception.
**lushr**

**Operation**
Logical shift right long

**Format**

| lushr |

**Forms**
lushr = 125 (0x7d)

**Operand**

..., value1, value2 ⇒

**Stack**

..., result

**Description**
The value1 must be of type long, and value2 must be of type int. The values are popped from the operand stack. A long result is calculated by shifting value1 right logically (with zero extension) by the amount indicated by the low 6 bits of value2. The result is pushed onto the operand stack.

**Notes**
If value1 is positive and s is value2 & 0x3f, the result is the same as that of value1 >> s; if value1 is negative, the result is equal to the value of the expression (value1 >> s) + (2L << ~s). The addition of the (2L << ~s) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 63, inclusive.
**lxor**

**Operation**  
Boolean XOR long

**Format**  

| lxor |

**Forms**  
lxor = 131 (0x83)

**Operand**  
..., value1, value2 ⇒

**Stack**  
..., result

**Description**  
Both value1 and value2 must be of type long. They are popped from the operand stack. A long result is calculated by taking the bitwise exclusive OR of value1 and value2. The result is pushed onto the operand stack.
monitorenter

**Operation**  Enter monitor for object

**Format**  

<table>
<thead>
<tr>
<th>Format</th>
<th>monitorenter</th>
</tr>
</thead>
</table>

**Forms**  

<table>
<thead>
<tr>
<th>Forms</th>
<th>monitorenter = 194 (0xc2)</th>
</tr>
</thead>
</table>

**Operand**  

<table>
<thead>
<tr>
<th>Operand</th>
<th>..., objectref ⇒</th>
</tr>
</thead>
</table>

**Stack**  

<table>
<thead>
<tr>
<th>Stack</th>
<th>...</th>
</tr>
</thead>
</table>

**Description**  
The `objectref` must be of type `reference`.

Each object has a monitor associated with it. The thread that executes `monitorenter` gains ownership of the monitor associated with `objectref`. If another thread already owns the monitor associated with `objectref`, the current thread waits until the object is unlocked, then tries again to gain ownership. If the current thread already owns the monitor associated with `objectref`, it increments a counter in the monitor indicating the number of times this thread has entered the monitor. If the monitor associated with `objectref` is not owned by any thread, the current thread becomes the owner of the monitor, setting the entry count of this monitor to 1.

**Runtime Exception**  
If `objectref` is `null`, `monitorenter` throws a `NullPointerException`.

**Notes**  
For detailed information about threads and monitors in the Java virtual machine, see Chapter 8, “Threads and Locks.”
A `monitorenter` instruction may be used with one or more `monitorexit` instructions to implement a synchronized statement in the Java programming language. The `monitorenter` and `monitorexit` instructions are not used in the implementation of synchronized methods, although they can be used to provide equivalent locking semantics; however, monitor entry on invocation of a synchronized method is handled implicitly by the Java virtual machine’s method invocation instructions. See Section 7.14 for more information on the use of the `monitorenter` and `monitorexit` instructions.

The association of a monitor with an object may be managed in various ways that are beyond the scope of this specification. For instance, the monitor may be allocated and deallocated at the same time as the object. Alternatively, it may be dynamically allocated at the time when a thread attempts to gain exclusive access to the object and freed at some later time when no thread remains in the monitor for the object.

The synchronization constructs of the Java programming language require support for operations on monitors besides entry and exit. These include waiting on a monitor (`Object.wait`) and notifying other threads waiting on a monitor (`Object.notifyAll` and `Object.notify`). These operations are supported in the standard package `java.lang` supplied with the Java virtual machine. No explicit support for these operations appears in the instruction set of the Java virtual machine.
**monitorexit**

**Operation**
Exit monitor for object

**Format**

```
monitorexit
```

**Forms**

`monitorexit = 195 (0xc3)`

**Operand**

`..., objectref ⇒`

**Stack**

`...`

**Description**

The `objectref` must be of type `reference`.

The current thread should be the owner of the monitor associated with the instance referenced by `objectref`. The thread decrements the counter indicating the number of times it has entered this monitor. If as a result the value of the counter becomes zero, the current thread releases the monitor. If the monitor associated with `objectref` becomes free, other threads that are waiting to acquire that monitor are allowed to attempt to do so.

**Runtime Exceptions**

If `objectref` is `null`, `monitorexit` throws a `NullPointerException`. Otherwise, if the current thread is not the owner of the monitor, `monitorexit` throws an `IllegalMonitorStateException`.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in Section 8.13 and if the second of those rules is violated by the execution of this `monitorexit` instruction, then `monitorexit` throws an `IllegalMonitorStateException`.

**Notes**

For detailed information about threads and monitors in the Java virtual machine, see Chapter 8, “Threads and Locks.”
One or more `monitorexit` instructions may be used with a `monitorenter` instruction to implement a synchronized statement in the Java programming language. The `monitorenter` and `monitorexit` instructions are not used in the implementation of synchronized methods, although they can be used to provide equivalent locking semantics.

The Java virtual machine supports exceptions thrown within synchronized methods and synchronized statements differently. Monitor exit on normal synchronized method completion is handled by the Java virtual machine’s return instructions. Monitor exit on abrupt synchronized method completion is handled implicitly by the Java virtual machine’s `athrow` instruction. When an exception is thrown from within a synchronized statement, exit from the monitor entered prior to the execution of the synchronized statement is achieved using the Java virtual machine’s exception handling mechanism. See Section 7.14 for more information on the use of the `monitorenter` and `monitorexit` instructions.
multianewarray

Operation  Create new multidimensional array

Format

<table>
<thead>
<tr>
<th>Operation</th>
<th>4-byte instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>multianewarray</td>
<td>197 (0xc5)</td>
</tr>
<tr>
<td>indexbyte1</td>
<td></td>
</tr>
<tr>
<td>indexbyte2</td>
<td></td>
</tr>
<tr>
<td>dimensions</td>
<td></td>
</tr>
</tbody>
</table>

Forms  multianewarray = 197 (0xc5)

Operand  ..., count1, [count2, ...] ⇒

Stack    ..., arrayref

Description  The dimensions operand is an unsigned byte that must be greater than or equal to 1. It represents the number of dimensions of the array to be created. The operand stack must contain dimensions values. Each such value represents the number of components in a dimension of the array to be created, must be of type int, and must be nonnegative. The count1 is the desired length in the first dimension, count2 in the second, etc.

All of the count values are popped off the operand stack. The unsigned indexbyte1 and indexbyte2 are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is (indexbyte1 << 8) | indexbyte2. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1). The resulting entry must be an array class type of dimensionality greater than or equal to dimensions.
multianewarray (cont.)

A new multidimensional array of the array type is allocated from the garbage-collected heap. If any count value is zero, no subsequent dimensions are allocated. The components of the array in the first dimension are initialized to subarrays of the type of the second dimension, and so on. The components of the last allocated dimension of the array are initialized to the default initial value for the type of the components (§2.5.1). A reference arrayref to the new array is pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in Section 5.4.3.1 can be thrown.

Otherwise, if the current class does not have permission to access the element type of the resolved array class, multianewarray throws an IllegalAccessError.

Runtime Exception

Otherwise, if any of the dimensions values on the operand stack are less than zero, the multianewarray instruction throws a Negative-ArraySizeException.

Notes

It may be more efficient to use newarray or anewarray when creating an array of a single dimension.

The array class referenced via the runtime constant pool may have more dimensions than the dimensions operand of the multianewarray instruction. In that case, only the first dimensions of the dimensions of the array are created.
**new**

**Operation**  Create new object

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td></td>
</tr>
<tr>
<td>indexbyte1</td>
<td></td>
</tr>
<tr>
<td>indexbyte2</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**  

new = 187 (0xbb)

**Operand Stack**  

... ⇒ ...

,..., objectref

**Description**  
The unsigned indexbyte1 and indexbyte2 are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is (indexbyte1 << 8) | indexbyte2. The runtime constant pool item at the index must be a symbolic reference to a class or interface type. The named class or interface type is resolved (§5.4.3.1) and should result in a class type. Memory for a new instance of that class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized to their default initial values (§2.5.1). The objectref, a reference to the instance, is pushed onto the operand stack.

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

**Linking Exceptions**  

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in Section 5.4.3.1 can be thrown.

Otherwise, if the symbolic reference to the class, array, or interface type resolves to an interface or is an abstract class, new throws an InstantiationException.
new (cont.)

Runtime Exception
Otherwise, if execution of this new instruction causes initialization of the referenced class, new may throw an Error as detailed in Section 2.17.5.

Note
The new instruction does not completely create a new instance; instance creation is not completed until an instance initialization method has been invoked on the uninitialized instance.
newarray

<table>
<thead>
<tr>
<th>Operation</th>
<th>Create new arrayhandler</th>
</tr>
</thead>
</table>

| Format    | \[
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>newarray</td>
<td></td>
</tr>
<tr>
<td>atype</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Forms</th>
<th>newarray = 188 (0xbc)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>[... \text{count} \rightarrow ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>[... \text{arrayref} ]</td>
</tr>
</tbody>
</table>

| Description | The \text{count} must be of type \text{int}. It is popped off the operand stack. The \text{count} represents the number of elements in the array to be created. |

The \text{atype} is a code that indicates the type of array to create. It must take one of the following values:

<table>
<thead>
<tr>
<th>Array Type</th>
<th>atype</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_BOOLEAN</td>
<td>4</td>
</tr>
<tr>
<td>T_CHAR</td>
<td>5</td>
</tr>
<tr>
<td>T_FLOAT</td>
<td>6</td>
</tr>
<tr>
<td>T_DOUBLE</td>
<td>7</td>
</tr>
<tr>
<td>T_BYTE</td>
<td>8</td>
</tr>
<tr>
<td>T_SHORT</td>
<td>9</td>
</tr>
<tr>
<td>T_INT</td>
<td>10</td>
</tr>
<tr>
<td>T_LONG</td>
<td>11</td>
</tr>
</tbody>
</table>

A new array whose components are of type \text{atype} and of length \text{count} is allocated from the garbage-collected heap. A reference \text{arrayref} to this new array object is pushed into the operand stack. Each of the elements of the new array is initialized to the default initial value for the type of the array (§2.5.1).
newarray (cont.)

Runtime Exception
If count is less than zero, newarray throws a NegativeArray-SizeException.

Notes
In Sun’s implementation of the Java virtual machine, arrays of type boolean (atype is T_BOOLEAN) are stored as arrays of 8-bit values and are manipulated using the baload and bastore instructions, instructions that also access arrays of type byte. Other implementations may implement packed boolean arrays; the baload and bastore instructions must still be used to access those arrays.
### nop

**Operation**  
Do nothing

**Format**

```
  nop
```

**Forms**  

\[ \text{nop} = 0 \ (0x0) \]

**Operand**  
No change

**Stack**  
No change

**Description**  
Do nothing.
Operation: Pop the top operand stack value

Format: \texttt{pop}

Forms: \texttt{pop} = 87 (0x57)

Operand: \ldots, value \Rightarrow

Stack: \ldots

Description: Pop the top value from the operand stack.

The \textit{pop} instruction must not be used unless \textit{value} is a value of a category 1 computational type (§3.11.1).
Operation  Pop the top one or two operand stack values

Format  

Forms  $pop2 = 88 \ (0x58)$

Operand Stack  

Form 1:

$\ldots, value2, value1 \Rightarrow$

$\ldots$

$\ldots$

where each of $value1$ and $value2$ is a value of a category 1 computational type ($\S3.11.1$).

Form 2:

$\ldots, value \Rightarrow$

$\ldots$

$\ldots$

where $value$ is a value of a category 2 computational type ($\S3.11.1$).

Description  Pop the top one or two values from the operand stack.
**putfield**

**Operation**  
Set field in object

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>putfield</td>
<td></td>
</tr>
<tr>
<td>indexbyte1</td>
<td></td>
</tr>
<tr>
<td>indexbyte2</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**  
`putfield = 181 (0xb5)`

**Operand**  
`..., objectref, value ⇒`

**Stack**  
`...`

**Description**  
The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The class of `objectref` must not be an array. If the field is protected (§4.6), and it is a member of a superclass of the current class, and the field is not declared in the same run-time package (§5.3) as the current class, then the class of `objectref` must be either the current class or a subclass of the current class.

The referenced field is resolved (§5.4.3.2). The type of a `value` stored by a `putfield` instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the `value` must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the `value` must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the `value` must be of a type that is assignment compatible (§2.6.7) with the field descriptor type. If the field is `final`, it should be declared in the current class, and the instruction should occur in an instance initialization method (`<init>`) method of the current class. Otherwise, an `IllegalAccessException` is thrown.
**putfield (cont.)**

The value and objectref are popped from the operand stack. The objectref must be of type reference. The value undergoes value set conversion (§3.8.3), resulting in value', and the referenced field in objectref is set to value'.

**Linking Exceptions**

During resolution of the symbolic reference to the field, any of the exceptions pertaining to field resolution documented in Section 5.4.3.2 can be thrown.

Otherwise, if the resolved field is a static field, *putfield* throws an IncompatibleClassChangeError.

Otherwise, if the field is final, it should be declared in the current class, and the instruction should occur in an instance initialization method (<init>) method of the current class. Otherwise, an IllegalAccessError is thrown.

**Runtime Exception**

Otherwise, if objectref is null, the *putfield* instruction throws a NullPointerException.
**putstatic**

**Operation**  
Set static field in class

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>putstatic</strong></td>
<td></td>
</tr>
<tr>
<td>indexbyte1</td>
<td></td>
</tr>
<tr>
<td>indexbyte2</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**

`putstatic = 179 (0xb3)`

**Operand**

`..., value ⇒`

**Stack**

`...`

**Description**

The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved (§5.4.3.2).

On successful resolution of the field the class or interface that declared the resolved field is initialized (§5.5) if that class or interface has not already been initialized.

The type of a `value` stored by a `putstatic` instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the `value` must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the `value` must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the `value` must be of a type that is assignment compatible (§2.6.7) with the field descriptor type. If the field is `final`, it should be declared in the current class, and the instruction should occur in the `<clinit>` method of the current class. Otherwise, an `IllegalAccessException` is thrown.
The *value* is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in *value’*. The class field is set to *value’*.

### Linking Exceptions
During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution documented in Section 5.4.3.2 can be thrown.

Otherwise, if the resolved field is not a static (class) field or an interface field, *putstatic* throws an *IncompatibleClassChangeError*.

Otherwise, if the field is *final*, it should be declared in the current class, and the instruction should occur in the <code>&lt;clinit&gt;</code> method of the current class. Otherwise, an *IllegalAccessError* is thrown.

### Runtime Exception
Otherwise, if execution of this *putstatic* instruction causes initialization of the referenced class or interface, *putstatic* may throw an *Error* as detailed in Section 2.17.5.

### Notes
A *putstatic* instruction may be used only to set the value of an interface field on the initialization of that field. Interface fields may be assigned to only once, on execution of an interface variable initialization expression when the interface is initialized (§2.17.4).
ret

**Operation**  Return from subroutine

**Format**  

<table>
<thead>
<tr>
<th>Operation</th>
<th>Return from subroutine</th>
</tr>
</thead>
</table>

**Forms**  

<table>
<thead>
<tr>
<th>Form</th>
<th>ret = 169 (0xa9)</th>
</tr>
</thead>
</table>

**Operand Stack**  

<table>
<thead>
<tr>
<th>Type</th>
<th>No change</th>
</tr>
</thead>
</table>

**Description**  
The index is an unsigned byte between 0 and 255, inclusive. The local variable at index in the current frame (§3.6) must contain a value of type returnAddress. The contents of the local variable are written into the Java virtual machine’s pc register, and execution continues there.

**Notes**  
The ret instruction is used with jsr or jsr_w instructions in the implementation of the finally clauses of the Java programming language (see Section 7.13, “Compiling finally”). Note that jsr pushes the address onto the operand stack and ret gets it out of a local variable. This asymmetry is intentional.

The ret instruction should not be confused with the return instruction. A return instruction returns control from a method to its invoker, without passing any value back to the invoker.

The ret opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
**return**

**Operation**  
Return void from method

**Format**

```
return
```

**Forms**  

```
return = 177 (0xb1)
```

**Operand**  
... ⇒

**Stack**  
[empty]

**Description**  
The current method must have return type void. If the current method is a synchronized method, the monitor acquired or reentered on invocation of the method is released or exited (respectively) as if by execution of a monitorexit instruction. If no exception is thrown, any values on the operand stack of the current frame (§3.6) are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

**Runtime Exceptions**  
If the current method is a synchronized method and the current thread is not the owner of the monitor acquired or reentered on invocation of the method, return throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monitorenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured use of locks described in Section 8.13 and if the first of those rules is violated during invocation of the current method, then return throws an IllegalMonitorStateException.
**saload**

**Operation**  
Load short from array

**Format**  
```
saload
```

**Forms**  
```
saload = 53 (0x35)
```

**Operand Stack**  
```
..., arrayref, index ⇒
```

**Description**  
The `arrayref` must be of type reference and must refer to an array whose components are of type short. The `index` must be of type int. Both `arrayref` and `index` are popped from the operand stack. The component of the array at `index` is retrieved and sign-extended to an int `value`. That `value` is pushed onto the operand stack.

**Runtime Exceptions**  
If `arrayref` is null, `saload` throws a NullPointerException.
Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `saload` instruction throws an ArrayIndexOutOfBoundsException.
**sastore**

**Operation**  
Store into short array

**Format**  
```
sastore
```

**Forms**  
sastore = 86 (0x56)

**Operand Stack**  
```
..., array, index, value ⇒
```

**Description**  
The `arrayref` must be of type reference and must refer to an array whose components are of type short. Both `index` and `value` must be of type int. The `arrayref`, `index`, and `value` are popped from the operand stack. The int `value` is truncated to a short and stored as the component of the array indexed by `index`.

**Runtime Exceptions**  
If `arrayref` is null, `sastore` throws a `NullPointerException`.

Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `sastore` instruction throws an `ArrayIndexOutOfBoundsException`.


**sipush**

**Operation**  Push short

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sipush</td>
<td></td>
</tr>
<tr>
<td>byte1</td>
<td></td>
</tr>
<tr>
<td>byte2</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**  sipush = 17 (0x11)

**Operand**  ... ⇒

**Stack**  ..., value

**Description**  The immediate unsigned byte1 and byte2 values are assembled into an intermediate short where the value of the short is (byte1 << 8) | byte2. The intermediate value is then sign-extended to an int value. That value is pushed onto the operand stack.
**swap**

**Operation**  Swap the top two operand stack values

**Format**

```
swap
```

**Forms**  

\[ \text{swap} = 95 \ (0x5f) \]

**Operand**  

\[ \ldots, \text{value2, value1} \Rightarrow \]

**Stack**  

\[ \ldots, \text{value1, value2} \]

**Description**  Swap the top two values on the operand stack.

The `swap` instruction must not be used unless `value1` and `value2` are both values of a category 1 computational type (§3.11.1).

**Notes**  The Java virtual machine does not provide an instruction implementing a swap on operands of category 2 computational types.
**tables**

**Operation**  Access jump table by index and jump

**Format**

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>tables**</td>
</tr>
<tr>
<td>&lt;0-3 byte pad&gt;</td>
</tr>
<tr>
<td>defaultbyte1</td>
</tr>
<tr>
<td>defaultbyte2</td>
</tr>
<tr>
<td>defaultbyte3</td>
</tr>
<tr>
<td>defaultbyte4</td>
</tr>
<tr>
<td>lowbyte1</td>
</tr>
<tr>
<td>lowbyte2</td>
</tr>
<tr>
<td>lowbyte3</td>
</tr>
<tr>
<td>lowbyte4</td>
</tr>
<tr>
<td>highbyte1</td>
</tr>
<tr>
<td>highbyte2</td>
</tr>
<tr>
<td>highbyte3</td>
</tr>
<tr>
<td>highbyte4</td>
</tr>
<tr>
<td>jump offsets...</td>
</tr>
</tbody>
</table>

**Forms**  tables** = 170 (0xaa)

**Operand**  ... index ⇒

**Stack**  ...

**Description**  A tables** is a variable-length instruction. Immediately after the tables** opcode, between 0 and 3 null bytes (zeroed bytes, not the null object) are inserted as padding. The number of null bytes is chosen so that the following byte begins at an address that is a multiple of 4 bytes from the start of the current method (the opcode of its first instruction). Immediately after the padding follow bytes constituting three signed 32-bit values: default, low, and high. Immediately following those bytes are bytes constituting a series of high − low + 1 signed 32-bit offsets. The value low must be less than or equal to high. The high − low + 1 signed 32-bit offsets are
treated as a 0-based jump table. Each of these signed 32-bit values is constructed as \((byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4\).
The `index` must be of type `int` and is popped from the operand stack. If `index` is less than `low` or `index` is greater than `high`, then a target address is calculated by adding `default` to the address of the opcode of this `tableswitch` instruction. Otherwise, the offset at position `index – low` of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this `tableswitch` instruction. Execution then continues at the target address.

The target address that can be calculated from each jump table offset, as well as the one that can be calculated from `default`, must be the address of an opcode of an instruction within the method that contains this `tableswitch` instruction.

**Notes**

The alignment required of the 4-byte operands of the `tableswitch` instruction guarantees 4-byte alignment of those operands if and only if the method that contains the `tableswitch` starts on a 4-byte boundary.
**wide**

**Operation**
Extend local variable index by additional bytes

**Format 1:**

<table>
<thead>
<tr>
<th>wide</th>
<th>&lt;opcode&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>indexbyte1</td>
</tr>
<tr>
<td></td>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

where `<opcode>` is one of `iload`, `fload`, `aload`, `lload`, `dload`, `istore`, `fstore`, `astore`, `lstore`, `dstore`, or `ret`

**Format 2:**

<table>
<thead>
<tr>
<th>wide</th>
<th>iinc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>indexbyte1</td>
</tr>
<tr>
<td></td>
<td>indexbyte2</td>
</tr>
<tr>
<td></td>
<td>constbyte1</td>
</tr>
<tr>
<td></td>
<td>constbyte2</td>
</tr>
</tbody>
</table>

**Forms**

`wide` = 196 (0xc4)

**Operand Stack**

**Description**
The `wide` instruction modifies the behavior of another instruction. It takes one of two formats, depending on the instruction being modified. The first form of the `wide` instruction modifies one of the instructions `iload`, `fload`, `aload`, `lload`, `dload`, `istore`, `fstore`, `astore`, `lstore`, `dstore`, or `ret`. The second form applies only to the `iinc` instruction.

In either case, the `wide` opcode itself is followed in the compiled code by the opcode of the instruction `wide` modifies. In either form, two unsigned bytes `indexbyte1` and `indexbyte2` follow the modified opcode and are assembled into a 16-bit unsigned index to a local variable in the current frame (§3.6), where the value of the index is
wide (cont.)

\[(\text{indexbyte1} \ll 8) | \text{indexbyte2}\]. The calculated index must be an index into the local variable array of the current frame. Where the wide instruction modifies an lload, dload, lstore, or dstore instruction, the index following the calculated index \((\text{index} + 1)\) must also be an index into the local variable array. In the second form, two immediate unsigned bytes \(\text{constbyte1}\) and \(\text{constbyte2}\) follow \(\text{indexbyte1}\) and \(\text{indexbyte2}\) in the code stream. Those bytes are also assembled into a signed 16-bit constant, where the constant is \((\text{constbyte1} \ll 8) | \text{constbyte2}\).

The widened bytecode operates as normal, except for the use of the wider index and, in the case of the second form, the larger increment range.

Notes

Although we say that wide “modifies the behavior of another instruction,” the wide instruction effectively treats the bytes constituting the modified instruction as operands, denaturing the embedded instruction in the process. In the case of a modified iinc instruction, one of the logical operands of the iinc is not even at the normal offset from the opcode. The embedded instruction must never be executed directly; its opcode must never be the target of any control transfer instruction.