| | |
|---|---|
| **Mnemonic:** | **aaload** |
| **Operation:** | Load **reference** from array. |
| **Opcode:** | 50 (0x32) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref***, ***index*** >>><br>After: …, ***value*** |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array whose components are of **reference** type. Furthermore, ***index*** must be of **int** type. Both ***arrayref*** and ***index*** pop from the operand stack. The **reference** ***value*** in the array component at ***index*** retrieves and pushes to the operand stack. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***arrayref*** is **null**.<br><br>This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array. |

**Mnemonic:** **aastore**

**Operation:** Store into **reference** array.

**Opcode:** 83 (0x53)

**Operands:** None

**Operand Stack:** Before: …, ***arrayref***, ***index***, ***value*** >>>
After: …

**Description:** ***arrayref*** must be of **reference** type and must refer to an array whose components are of **reference** type. Furthermore, ***index*** must be of **int** type and ***value*** must be of **reference** type. ***arrayref***, ***index***, and ***value*** pop from the operand stack. The **reference** ***value*** stores as the array component at ***index***.

***value***'s type must be assignment compatible with the type of the ***arrayref***-referenced array's components. Assignment of a value of reference type *S* (source) to a variable of reference type *T* (target) is allowed only when type *S* supports all the operations defined on type *T*. The detailed rules:

1) If *S* is a class type, then:

1.1) If *T* is a class type, then *S* must be the same class as *T*, or *S* must be a subclass of *T*.

1.2) If *T* is an interface type, *S* must implement interface *T*.

2) If *S* is an interface type, then:

2.1) If *T* is a class type, then *T* must be **Object**.

2.2) If *T* is an interface type, then *T* must be the same interface as *S* or a superinterface of *S*.

3) If *S* is an array type, namely *SC*[] (an array of components of type *SC*), then:

3.1) If *T* is a class type, then *T* must be **Object**.

3.2) If *T* is an array type *TC*[] (an array of components of type *TC*), then one of the following must be true:

3.2.1) *TC* and *SC* are the same primitive type.

3.2.2) *TC* and *SC* are reference types, and type *SC* is assignable to *TC* by these runtime rules.

3.3) If *T* is an interface type, *T* must be one of the interfaces implemented by arrays.

**Runtime Exceptions:**

This instruction throws a **NullPointerException** if ***arrayref*** is **null**.

This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array.

This instruction throws an **ArrayStoreException** if **arrayref** is not **null** and ***value***'s actual type is not assignment compatible with the actual type of the array's components.

**Mnemonic:** **aconst_null**

**Operation:** Push **null**.

**Opcode:** 1 (0x01)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **null**

**Description:** Push the **null** object **reference** to the operand stack.

**Notes:** A specific value for **null** is not mandated by the JVM specification.

**Mnemonic:** **aload**

**Operation:** Load **reference** from local variable.

**Opcode:** 25 (0x19)

**Operands:** ***index***

**Operand Stack:** Before: … >>>
After: …, ***objectref***

**Description:** ***index*** is an unsigned byte that must be an index into the current stack frame's local variable array. Furthermore, the local variable at ***index*** must contain a **reference**. The ***objectref*** in the local variable at ***index*** pushes to the operand stack.

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

This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **aload_0**

**Operation:** Load **reference** from local variable 0.

**Opcode:** 42 (0x2A)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***objectref***

**Description:** 0 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 0 must contain a **reference**. The ***objectref*** in the local variable at 0 pushes to the operand stack.

**Notes:** This instruction cannot be used to load a value of type **returnAddress** from a local variable to the operand stack. This asymmetry with **astore_0** is intentional.

This instruction is the same as **aload** (with a 0 index operand), except that the 0 in **aload_0** is implicit.

**Mnemonic:** **aload_1**

**Operation:** Load **reference** from local variable 1.

**Opcode:** 43 (0x2B)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***objectref***

**Description:** 1 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 1 must contain a **reference**. The ***objectref*** in the local variable at 1 pushes to the operand stack.

**Notes:** This instruction cannot be used to load a value of type **returnAddress** from a local variable to the operand stack. This asymmetry with **astore_1** is intentional.

This instruction is the same as **aload** (with a 1 index operand), except that the 1 in **aload_1** is implicit.

**Mnemonic:** **aload_2**

**Operation:** Load **reference** from local variable 2.

**Opcode:** 44 (0x2C)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***objectref***

**Description:** 2 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 2 must contain a **reference**. The ***objectref*** in the local variable at 2 pushes to the operand stack.

**Notes:** This instruction cannot be used to load a value of type **returnAddress** from a local variable to the operand stack. This asymmetry with **astore_2** is intentional.

This instruction is the same as **aload** (with a 2 index operand), except that the 2 in **aload_2** is implicit.

**Mnemonic:** **aload_3**

**Operation:** Load **reference** from local variable 3.

**Opcode:** 45 (0x2D)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***objectref***

**Description:** 3 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 3 must contain a **reference**. The ***objectref*** in the local variable at 3 pushes to the operand stack.

**Notes:** This instruction cannot be used to load a value of type **returnAddress** from a local variable to the operand stack. This asymmetry with **astore_3** is intentional.

This instruction is the same as **aload** (with a 3 index operand), except that the 3 in **aload_3** is implicit.

| | |
|---|---|
| **Mnemonic:** | **anewarray** |
| **Operation:** | Create new array of **reference**. |
| **Opcode:** | 189 (0xBD) |
| **Operands:** | ***indexbyte1***, ***indexbyte2*** |
| **Operand Stack:** | Before: …, ***count*** >>><br>After: …, ***arrayref*** |
| **Description:** | ***count***, which must be of **int** type, pops from the operand stack and represents the number of array components to be created. The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index value is (***indexbyte1***<<8)\|***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to a class, array, or interface type. The named type resolves, a new array with that type's components and a ***count*** length creates from the garbage-collected heap, and an ***arrayref*** **reference** to this new array object pushes to the operand stack. All array components initialize to **null**, which is the default value for **reference** types. |
| **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 of the JVM specification can be thrown. |
| **Runtime Exceptions:** | This instruction throws a **NegativeArraySizeException** if ***count*** is less than zero. |
| **Notes:** | Use this instruction to create either a one-dimensional array of object references or part of a multidimensional array. |

**Mnemonic:** **areturn**

**Operation:** Return **reference** from method.

**Opcode:** 176 (0xB0)

**Operands:** None

**Operand Stack:** Before: …, ***objectref*** >>>
After: [empty]

**Description:** ***objectref*** must be of **reference** type and must refer to an object whose type is assignment compatible with the type represented by the current method's return descriptor. If the current method is **synchronized**, the monitor acquired or reentered on that method's invocation is released or exited (respectively) as if by executing the **monitorexit** instruction. If no exception is thrown, ***objectref*** pops from the current stack frame's operand stack and pushes to the operand stack of the invoker's stack frame. Any other values on the current method's operand stack discard.

The interpreter returns control to the current method's invoker, reinstating the invoker's stack frame as the current stack frame.

**Runtime Exceptions:** This instruction throws an **IllegalMonitorStateException** 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.

This instruction also throws an **IllegalMonitorStateException** if the JVM implementation enforces the rules on structured lock usage, and the number of lock operations performed by a thread on a lock does not match the number of unlock operations by that thread on that lock (whether the method invocation completes normally or abruptly).

| | |
|---|---|
| **Mnemonic:** | **arraylength** |
| **Operation:** | Get length of array. |
| **Opcode:** | 190 (0xBE) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref*** >>><br>After: …, ***length*** |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array. It pops from the operand stack. The ***length*** of the array it references is determined, and pushes to the operand stack as an **int**. |
| **Runtime Exceptions:** | This instruction throws an **NullPointerException** if ***arrayref*** is **null**. |

**Mnemonic:** **astore**

**Operation:** Store **reference** into local variable.

**Opcode:** 58 (0x3A)

**Operands:** ***index***

**Operand Stack:** Before: …, ***objectref*** >>>
After: …

**Description:** ***index*** is an unsigned byte that must be an index into the current stack frame's local variable array. Furthermore, the ***objectref*** on the top of the operand stack must be of **returnAddress** type or of **reference** type. It pops from the operand stack and the value of the local variable at ***index*** sets to ***objectref***.

**Notes:** The Java programming language uses this instruction with an ***objectref*** of type **returnAddress** when implementing **finally** clauses.

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

This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **astore_0**

**Operation:** Store **reference** into local variable 0.

**Opcode:** 75 (0x4B)

**Operands:** None

**Operand Stack:** Before: …, ***objectref*** >>>
After: …

**Description:** 0 must be an index into the current stack frame's local variable array. Furthermore, the ***objectref*** on the top of the operand stack must be of **returnAddress** type or of **reference** type. It pops from the operand stack and the value of the local variable at 0 sets to ***objectref***.

**Notes:** The Java programming language uses this instruction with an ***objectref*** of type **returnAddress** when implementing **finally** clauses.

The **aload_0** instruction cannot be used to load a value of type **returnAddress** from a local variable to the operand stack. This asymmetry with **astore_0** is intentional.

This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **astore_1**

**Operation:** Store **reference** into local variable 1.

**Opcode:** 76 (0x4C)

**Operands:** None

**Operand Stack:** Before: …, ***objectref*** >>>
After: …

**Description:** 1 must be an index into the current stack frame's local variable array. Furthermore, the ***objectref*** on the top of the operand stack must be of **returnAddress** type or of **reference** type. It pops from the operand stack and the value of the local variable at 1 sets to ***objectref***.

**Notes:** The Java programming language uses this instruction with an ***objectref*** of type **returnAddress** when implementing **finally** clauses.

The **aload_1** instruction cannot be used to load a value of type **returnAddress** from a local variable to the operand stack. This asymmetry with **astore_1** is intentional.

This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **astore_2**

**Operation:** Store **reference** into local variable 2.

**Opcode:** 77 (0x4D)

**Operands:** None

**Operand Stack:** Before: …, ***objectref*** >>>
After: …

**Description:** 2 must be an index into the current stack frame's local variable array. Furthermore, the ***objectref*** on the top of the operand stack must be of **returnAddress** type or of **reference** type. It pops from the operand stack and the value of the local variable at 2 sets to ***objectref***.

**Notes:** The Java programming language uses this instruction with an ***objectref*** of type **returnAddress** when implementing **finally** clauses.

The **aload_2** instruction cannot be used to load a value of type **returnAddress** from a local variable to the operand stack. This asymmetry with **astore_2** is intentional.

This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **astore_3**

**Operation:** Store **reference** into local variable 3.

**Opcode:** 78 (0x4E)

**Operands:** None

**Operand Stack:** Before: …, ***objectref*** >>>
After: …

**Description:** 3 must be an index into the current stack frame's local variable array. Furthermore, the ***objectref*** on the top of the operand stack must be of **returnAddress** type or of **reference** type. It pops from the operand stack and the value of the local variable at 3 sets to ***objectref***.

**Notes:** The Java programming language uses this instruction with an ***objectref*** of type **returnAddress** when implementing **finally** clauses.

The **aload_3** instruction cannot be used to load a value of type **returnAddress** from a local variable to the operand stack. This asymmetry with **astore_3** is intentional.

This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **athrow**

**Operation:** Throw exception or error.

**Opcode:** 191 (0xBF)

**Operands:** None

**Operand Stack:** Before: …, ***objectref*** >>>
After: ***objectref***

**Description:** ***objectref*** must be of **reference** type and must refer to an object that is an instance of class **Throwable** or a **Throwable** subclass. It pops from the operand stack, and then is thrown by searching the current method for the first exception handler that matches ***objectref***'s class.

If an exception handler that matches ***objectref*** is found, it contains the location of the code that handles this exception. Register **pc** resets to that location, the current stack frame's operand stack clears, ***objectref*** pushes back to the operand stack, and execution continues.

If no matching exception handler is found in the current stack frame, the stack frame pops. If the current stack frame represents an invocation of a **synchronized** method, the monitor acquired or reentered on the method's invocation releases or exits (respectively) as if by executing the **monitorexit** instruction. The stack frame of its invoker is reinstated, if the stack frame exists, and the ***objectref*** rethrows. If no such stack frame exists, the current thread exits.

**Runtime Exceptions:** This instruction throws a **NullPointerException** instead of ***objectref*** if ***objectref*** is **null**.

This instruction throws an **IllegalMonitorStateException** 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.

This instruction also throws an **IllegalMonitorStateException** if the JVM implementation enforces the rules on structured lock usage, and the number of lock operations performed by a thread on a lock does not match the number of unlock operations by that thread on that lock (whether the method invocation completes normally or abruptly).

**Notes:** If a handler for the thrown exception or error matches in the current method, **athrow** discards all the values on the operand stack and then pushes the thrown object to the operand stack. However, if no handler matches in the current method and the exception throws farther up the method invocation chain, the operand stack of the method (if any) that handles the exception clears and ***objectref*** pushes to the empty operand stack. All intervening stack frames from the method that threw the exception up to (but not including) the method that handles the exception discard.

**Mnemonic:** **baload**

**Operation:** Load **byte** or **boolean** from array.

**Opcode:** 51 (0x33)

**Operands:** None

**Operand Stack:** Before: …, ***arrayref***, ***index*** >>>
After: …, ***value***

**Description:** ***arrayref*** must be of **reference** type and must refer to an array whose components are of **byte** or **boolean** type. Furthermore, ***index*** must be of **int** type. Both ***arrayref*** and ***index*** pop from the operand stack. If the array's components are of type **byte**, the array component at ***index*** retrieves and sign-extends to an **int** ***value***. If the array's components are of type **boolean**, the array component at ***index*** retrieves and zero-extends to an **int** ***value***. In either case, the resulting ***value*** pushes to the operand stack.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***arrayref*** is **null**.

This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array.

**Notes:** Sun's JVM implementation implements **boolean** arrays as arrays of 8-bit values.

**Mnemonic:** **bastore**

**Operation:** Store into **byte** or **boolean** array.

**Opcode:** 84 (0x54)

**Operands:** None

**Operand Stack:** Before: …, ***arrayref***, ***index***, ***value*** >>>
After: …

**Description:** ***arrayref*** must be of **reference** type and must refer to an array whose components are of **byte** or **boolean** type. Furthermore, ***index*** and ***value*** must both be of **int** type. ***arrayref***, ***index***, and ***value*** pop from the operand stack. If the array's components are of type **byte**, the **int** ***value*** truncates to a **byte** and stores as the array component at ***index***. If the array's components are of type **boolean**, the **int** ***value*** truncates to its low order bit and then zero-extends to the storage size for components of **boolean** arrays used by the JVM implementation. The result stores as the array component at ***index***.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***arrayref*** is **null**.

This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array.

**Notes:** Sun's JVM implementation implements **boolean** arrays as arrays of 8-bit values.

**Mnemonic:** **bipush**

**Operation:** Push **byte**.

**Opcode:** 16 (0x10)

**Operands:** ***byte***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** Sign-extend the immediate ***byte*** to an **int** ***value***. Push that ***value*** to the operand stack.

**Mnemonic:** **caload**

**Operation:** Load **char** from array.

**Opcode:** 52 (0x34)

**Operands:** None

**Operand Stack:** Before: …, ***arrayref***, ***index*** >>>
After: …, ***value***

**Description:** ***arrayref*** must be of **reference** type and must refer to an array whose components are of **char** type. Furthermore, ***index*** must be of **int** type. Both ***arrayref*** and ***index*** pop from the operand stack. The array component at ***index*** retrieves and zero-extends to an **int** ***value***, which pushes to the operand stack.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***arrayref*** is **null**.

This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array.

| | |
|---|---|
| **Mnemonic:** | **castore** |
| **Operation:** | Store into **char** array. |
| **Opcode:** | 85 (0x55) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref***, ***index***, ***value*** >>><br>After: … |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array whose components are of **char** type. Furthermore, ***index*** and ***value*** must both be of **int** type. ***arrayref***, ***index***, and ***value*** pop from the operand stack. The **int** ***value*** truncates to a **char**, which stores as the array component at ***index***. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***arrayref*** is **null**.<br><br>This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array. |

**Mnemonic:** **checkcast**

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

**Opcode:** 192 (0xC0)

**Operands:** ***indexbyte1***, ***indexbyte2***

**Operand Stack:** Before: …, ***objectref*** >>>
After: …, ***objectref***

**Description:** ***objectref*** must be of **reference** type. The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, 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 then resolves.

If ***objectref*** is **null** or can be cast to the resolved class, array, or interface type, the operand stack is unchanged. Otherwise, this instruction throws a **ClassCastException**.

The following rules determine whether a non-**null** ***objectref*** can be cast to the resolved type: If *S* is the class of the object referred to be ***objectref*** and *T* is the resolved class, array, or interface type, this instruction determines whether ***objectref*** can be cast to type *T* as follows:

1) If *S* is an ordinary (nonarray) class, then:

1.1) If *T* is a class type, then *S* must be the same class as *T*, or a subclass of *T*.

1.2) If *T* is an interface type, then *S* must implement interface *T*.

2) If *S* is an interface type, then:

2.1) If *T* is a class type, then *T* must be **Object**.

2.2) If *T* is an interface type, then *T* must be the same interface as *S* or a superinterface of *S*.

3) If *S* is a class representing the array type *SC*[] (an array of components of type *SC*), then:

3.1) If *T* is a class type, then *T* must be **Object**.

3.2) If *T* is an array type *TC*[] (an array of components of type *TC*), then one of the following must be true:

3.2.1) *TC* and *SC* are the same primitive type.

3.2.2) *TC* and *SC* are reference types, and type *SC* can be cast to *TC* by recursive

application of these rules.

3.3) If *T* is an interface type, *T* must be one of the interfaces implemented by arrays.

**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 of the JVM specification can be thrown.

**Runtime Exceptions:** This instruction throws a **ClassCastException** if ***objectref*** cannot be cast to the resolved class, array, or interface type.

**Notes:** This instruction is very similar to **instanceof**. However, **checkcast** differs in its treatment of **null**, its behavior when its test fails (**checkcast** throws an exception, whereas **instanceof** pushes a result code), and its effect on the operand stack.

**Mnemonic:** **d2f**

**Operation:** Convert **double** to **float**.

**Opcode:** 144 (0x90)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** The ***value*** on the top of the operand stack must be of **double** type. It pops from that stack and undergoes value set conversion, resulting in ***value'***. Then ***value'*** converts to a **float** ***result*** using IEEE 754 round to nearest mode. The ***result*** pushes to the operand stack.

Where a **d2f** instruction is FP-strict, the result of the conversion always rounds to the nearest representable value in the float value set.

Where a **d2f** instruction is not FP-strict, the result of the conversion may be taken from the float-extended-exponent value set; it is not necessarily rounded to the nearest representable value in the float value set.

A finite ***value'*** too small to represent as a **float** converts to a zero of the same sign; a finite ***value'*** too large to represent as a **float** converts to an infinity of the same sign. A **double** NaN converts to a **float** NaN.

**Notes:** This instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of ***value'*** and may also lose precision.

**Mnemonic:** **d2i**

**Operation:** Convert **double** to **int**.

**Opcode:** 142 (0x8E)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** The ***value*** on the top of the operand stack must be of **double** type. It pops from that stack and undergoes value set conversion, resulting in ***value'***. Then ***value'*** converts to an **int** ***result***. The ***result*** pushes to the operand stack:

1) If the ***value'*** is NaN, the ***result*** of the conversion is an **int** 0.

2) Otherwise, if the ***value'*** is not an infinity, it rounds to an integer value ***v***, rounding towards zero using IEEE 754 round towards zero mode. If this integer value ***v*** can represent as an **int**, the ***result*** is the **int** value ***v***.

3) 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:** This instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of ***value'*** and may also lose precision.

**Mnemonic:** **d2l**

**Operation:** Convert **double** to **long**.

**Opcode:** 143 (0x8F)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** The ***value*** on the top of the operand stack must be of **double** type. It pops from that stack and undergoes value set conversion, resulting in ***value'***. Then ***value'*** converts to a **long** ***result***. The ***result*** pushes to the operand stack:

1) If the ***value'*** is NaN, the ***result*** of the conversion is a **long** 0.

2) Otherwise, if the ***value'*** is not an infinity, it rounds to an integer value ***v***, rounding towards zero using IEEE 754 round towards zero mode. If this integer value ***v*** can represent as a **long**, the ***result*** is the **long** value ***v***.

3) 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:** This instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of ***value'*** and may also lose precision.

**Mnemonic:** **dadd**

**Operation:** Add **double**.

**Opcode:** 99 (0x63)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **double** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **double** ***result*** is ***value1'***+***value2'***, which pushes to the operand stack.

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

1) If either ***value1'*** or ***value2'*** is NaN, the result is NaN.

2) The sum of two infinities of opposite sign is NaN.

3) The sum of two infinities of the same sign is the infinity of that sign.

4) The sum of an infinity and any finite value is equal to the infinity.

5) The sum of two zeroes of opposite sign is positive zero.

6) The sum of two zeroes of the same sign is the zero of that sign.

7) The sum of a zero and a nonzero finite value is equal to the nonzero value.

8) The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.

9) In the remaining cases, where neither ***value1'*** nor ***value2'*** is an infinity, zero, or NaN and the values have the same sign or have different magnitudes, the sum computes and rounds to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a **double**, the operation is said to overflow; the result is an infinity of appropriate sign. If the magnitude is too small to represent as a **double**, the operation is said to underflow; the result is a zero of appropriate sign.

The JVM requires support of gradual underflow, as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of this instruction never throws a runtime exception.

**Mnemonic:** **daload**

**Operation:** Load **double** from array.

**Opcode:** 49 (0x31)

**Operands:** None

**Operand Stack:** Before: …, ***arrayref***, ***index*** >>>
After: …, ***value***

**Description:** ***arrayref*** must be of **reference** type and must refer to an array whose components are of **double** type. Furthermore, ***index*** must be of **int** type. Both ***arrayref*** and ***index*** pop from the operand stack. The **double** ***value*** in the array component at ***index*** retrieves and pushes to the operand stack.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***arrayref*** is **null**.

This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array.

| | |
|---|---|
| **Mnemonic:** | **dastore** |
| **Operation:** | Store into **double** array. |
| **Opcode:** | 82 (0x52) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref***, ***index***, ***value*** >>><br>After: … |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array whose components are of **double** type. Furthermore, ***index*** must be of **int** type and ***value*** must be of **double** type. ***arrayref***, ***index***, and ***value*** pop from the operand stack. The **double** ***value*** undergoes value set conversion, resulting in ***value'***, which stores as the array component at ***index***. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***arrayref*** is **null**.<br><br>This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array. |

**Mnemonic:** **dcmpg**

**Operation:** Compare **double**s: which is greater.

**Opcode:** 152 (0x98)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **double** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. A floating-point comparison is then performed:

1) If ***value1'*** is greater than ***value2'***, **int** value 1 pushes to the operand stack.

2) If ***value1'*** is equal to ***value2'***, **int** value 0 pushes to the operand stack.

3) If ***value1'*** is less than ***value2'***, **int** value -1 pushes to the operand stack.

4) If at least one of ***value1'*** or ***value2'*** is NaN, **int** value 1 pushes to the operand stack.

IEEE 754 rules for floating-point comparison are observed. All values other than NaN are ordered, with negative infinity being less than all finite values, and positive infinity being greater than all finite values. Positive zero and negative zero are considered to be equal.

**Notes:** **dcmpg** and **dcmpl** differ only in their treatment of a comparison involving NaN.

**Mnemonic:** **dcmpl**

**Operation:** Compare **double**s: which is lesser.

**Opcode:** 151 (0x97)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **double** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. A floating-point comparison is then performed:

1) If ***value1'*** is greater than ***value2'***, **int** value 1 pushes to the operand stack.

2) If ***value1'*** is equal to ***value2'***, **int** value 0 pushes to the operand stack.

3) If ***value1'*** is less than ***value2'***, **int** value -1 pushes to the operand stack.

4) If at least one of ***value1'*** or ***value2'*** is NaN, **int** value -1 pushes to the operand stack.

IEEE 754 rules for floating-point comparison are observed. All values other than NaN are ordered, with negative infinity being less than all finite values, and positive infinity being greater than all finite values. Positive zero and negative zero are considered to be equal.

**Notes:** **dcmpg** and **dcmpl** differ only in their treatment of a comparison involving NaN.

**Mnemonic:** **dconst_0**

**Operation:** Push **double 0.0**.

**Opcode:** 14 (0x0E)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **0.0**

**Description:** Push the **double** constant 0.0 to the operand stack.

**Mnemonic:** **dconst_1**

**Operation:** Push **double 1.0**.

**Opcode:** 15 (0x0F)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **1.0**

**Description:** Push the **double** constant 1.0 to the operand stack.

**Mnemonic:** **ddiv**

**Operation:** Divide **double**.

**Opcode:** 111 (0x6F)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **double** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **double** ***result*** from ***value1'***/***value2'*** pushes to the operand stack.

The following IEEE arithmetic rules govern this instruction:

1) If either ***value1'*** or ***value2'*** is NaN, the result is NaN.

2) If neither ***value1'*** nor ***value2'*** is NaN, the result's sign is positive (if both values have the same sign) or negative (if the values have different signs).

3) Division of positive/negative infinity by positive/negative infinity yields NaN.

4) Division of positive/negative infinity by a finite value results in a signed infinity (where the sign is based on Rule 2).

5) Division of a finite value by positive/negative infinity results in a signed zero (where the sign is based on Rule 2).

6) Division of positive/negative zero by positive/negative zero yields NaN. Division of zero by any other finite value results in a signed zero (where the sign is based on Rule 2).

7) Division of a nonzero finite value by a zero results in a signed infinity (where the sign is based on Rule 2).

8) In remaining cases, where neither ***value1'*** nor ***value2'*** is an infinity, a zero, or NaN, the quotient computes and rounds to the nearest **double** using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a **double**, the operation is said to overflow and the result is an infinity of the appropriate sign. If the magnitude is too small to represent as a **double**, the operation is said to underflow and the result is a zero of the appropriate sign.

The JVM requires support of gradual underflow as defined by IEEE 754. This instruction never throws an exception, even though overflow, underflow, division by zero, or loss of precision may occur.

**Mnemonic:** **dload**

**Operation:** Load **double** from local variable.

**Opcode:** 24 (0x18)

**Operands:** ***index***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** ***index*** is an unsigned byte. Both ***index*** and ***index***+1 must be indices into the current stack frame's local variable array. Furthermore, the local variable at ***index*** must contain a **double**. The ***value*** of the local variable at ***index*** pushes to the operand stack.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **dload_0**

**Operation:** Load **double** from local variable 0.

**Opcode:** 38 (0x26)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 0 and 1 must be indices into the current stack frame's local variable array. Furthermore, the local variable at 0 must contain a **double**. The ***value*** of the local variable at 0 pushes to the operand stack.

**Notes:** This instruction is the same as **dload** (with a 0 index operand), except that the 0 in **dload_0** is implicit.

**Mnemonic:** **dload_1**

**Operation:** Load **double** from local variable 1.

**Opcode:** 39 (0x27)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 1 and 2 must be indices into the current stack frame's local variable array. Furthermore, the local variable at 1 must contain a **double**. The ***value*** of the local variable at 1 pushes to the operand stack.

**Notes:** This instruction is the same as **dload** (with a 1 index operand), except that the 1 in **dload_1** is implicit.

**Mnemonic:** **dload_2**

**Operation:** Load **double** from local variable 2.

**Opcode:** 40 (0x28)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 2 and 3 must be indices into the current stack frame's local variable array. Furthermore, the local variable at 2 must contain a **double**. The ***value*** of the local variable at 2 pushes to the operand stack.

**Notes:** This instruction is the same as **dload** (with a 2 index operand), except that the 2 in **dload_2** is implicit.

**Mnemonic:** **dload_3**

**Operation:** Load **double** from local variable 3.

**Opcode:** 41 (0x29)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 3 and 4 must be indices into the current stack frame's local variable array. Furthermore, the local variable at 3 must contain a **double**. The ***value*** of the local variable at 3 pushes to the operand stack.

**Notes:** This instruction is the same as **dload** (with a 3 index operand), except that the 3 in **dload_3** is implicit.

**Mnemonic:** **dmul**

**Operation:** Multiply **double**.

**Opcode:** 107 (0x6B)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **double** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **double** ***result*** from ***value1'*******value2'*** pushes to the operand stack.

The following IEEE arithmetic rules govern this instruction:

1) If either ***value1'*** or ***value2'*** is NaN, the result is NaN.

2) If neither ***value1'*** nor ***value2'*** is NaN, the result's sign is positive (if both values have the same sign) or negative (if the values have different signs).

3) Multiplication of positive/negative infinity by positive/negative zero yields NaN.

4) Multiplication of positive/negative infinity by a finite value results in a signed infinity (where the sign is based on Rule 2).

5) In remaining cases, where neither an infinity nor NaN is involved, the product computes and rounds to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a **double**, the operation is said to overflow and the result is an infinity of the appropriate sign. If the magnitude is too small to represent as a **double**, the operation is said to underflow and the result is a zero of the appropriate sign.

The JVM requires support of gradual underflow as defined by IEEE 754. This instruction never throws an exception, even though overflow, underflow, or loss of precision may occur.

**Mnemonic:** **dneg**

**Operation:** Negate **double**.

**Opcode:** 119 (0x77)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value*** must be of **double** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. The **double** ***result*** is the arithmetic negation of ***value'***, which pushes to the operand stack.

Negation is not the same as subtraction from zero, for **double** values. If ***x*** is +0.0 then 0.0-***x*** equals +0.0, but –***x*** equals –0.0. (Unary minus inverts the sign of a **double**.)

There are three special cases:

1) If ***value*** is NaN, the result is NaN. (NaN has no sign.)

2) If ***value*** is positive/negative infinity, the result is the infinity of the opposite sign.

3) If ***value*** is positive/negative zero, the result is the zero of the opposite sign.

**Mnemonic:** **drem**

**Operation:** Remainder **double**.

**Opcode:** 115 (0x73)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **double** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **double** ***result*** calculates and pushes to the operand stack.

This instruction behaves in a manner similar to that of the **irem** and **lrem** JVM instructions.

The following rules govern this instruction:

1) If either ***value1'*** or ***value2'*** is NaN, the result is NaN.

2) If neither ***value1'*** nor ***value2'*** is NaN, the result's sign equals the dividend's sign.

3) If the dividend is positive/negative infinity or the divisor is positive/negative zero (or both), the result is NaN.

4) If the dividend is finite and the divisor is positive/negative infinity, the result equals the dividend.

5) If the dividend is positive/negative zero and the divisor is finite, the result equals the dividend.

6) In remaining cases, where neither ***value1*** nor ***value2*** is a positive/negative infinity, a positive/negative zero, nor NaN, the floating-point remainder ***result*** from a dividend ***value1'*** and a divisor ***value2'*** is defined by the mathematical relation ***result***=***value1'***–(***value2'*******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'***.

Even though division by zero may occur, this instruction never throws a runtime exception. Overflow, underflow, or precision loss cannot occur.

**Notes:** This instruction's result is not the same as that of the IEEE 754 remainder operation. Use the **Math.IEEEremainder** library routine to compute that operation.

**Mnemonic:** **dreturn**

**Operation:** Return **double** from method.

**Opcode:** 175 (0xAF)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: [empty]

**Description:** Both the current method's return type and ***value***'s type must be **double**. If the current method is **synchronized**, the monitor acquired or reentered on that method's invocation is released or exited (respectively) as if by executing the **monitorexit** instruction. If no exception is thrown, ***value*** pops from the current stack frame's operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** pushes to the operand stack of the invoker's stack frame. Any other values on the current method's operand stack discard.

The interpreter returns control to the current method's invoker, reinstating the invoker's stack frame as the current stack frame.

**Runtime Exceptions:** This instruction throws an **IllegalMonitorStateException** 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.

This instruction also throws an **IllegalMonitorStateException** if the JVM implementation enforces the rules on structured lock usage, and the number of lock operations performed by a thread on a lock does not match the number of unlock operations by that thread on that lock (whether the method invocation completes normally or abruptly).

**Mnemonic:** **dstore**

**Operation:** Store **double** into local variable.

**Opcode:** 57 (0x39)

**Operands:** ***index***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***index*** is an unsigned byte. Both ***index*** and ***index***+1 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **double** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variables at ***index*** and ***index***+1.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **dstore_0**

**Operation:** Store **double** into local variable 0.

**Opcode:** 71 (0x47)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 0 and 1 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **double** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variables at 0 and 1.

**Notes:** This instruction is the same as **dstore** (with a 0 index operand), except that the 0 in **dstore_0** is implicit.

**Mnemonic:** **dstore_1**

**Operation:** Store **double** into local variable 1.

**Opcode:** 72 (0x48)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 1 and 2 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **double** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variables at 1 and 2.

**Notes:** This instruction is the same as **dstore** (with a 1 index operand), except that the 1 in **dstore_1** is implicit.

**Mnemonic:** **dstore_2**

**Operation:** Store **double** into local variable 2.

**Opcode:** 73 (0x49)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 2 and 3 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **double** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variables at 2 and 3.

**Notes:** This instruction is the same as **dstore** (with a 2 index operand), except that the 2 in **dstore_2** is implicit.

**Mnemonic:** **dstore_3**

**Operation:** Store **double** into local variable 3.

**Opcode:** 74 (0x4A)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 3 and 4 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **double** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variables at 3 and 4.

**Notes:** This instruction is the same as **dstore** (with a 3 index operand), except that the 3 in **dstore_3** is implicit.

**Mnemonic:** **dsub**

**Operation:** Subtract **double**.

**Opcode:** 103 (0x67)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **double** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **double** ***result*** is ***value1'-value2'***, which pushes to the operand stack.

For **double** subtraction, it is always the case that ***a-b*** results in the same value as ***a***+(-***b***). However, subtraction from zero is not the same as negation: if ***x*** is +0.0, 0.0-***x*** equals +0.0, but -***x*** equals -0.0.

The JVM requires support of gradual underflow, as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of this instruction never throws a runtime exception.

**Mnemonic:** **dup**

**Operation:** Duplicate the top operand stack value.

**Opcode:** 89 (0x59)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***value***, ***value***

**Description:** Duplicate the top value on the operand stack and push the duplicated value to that stack.

This instruction must not be used if ***value*** is of type **double** or **long**.

**Mnemonic:** **dup_x1**

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

**Opcode:** 90 (0x5A)

**Operands:** None

**Operand Stack:** Before: …, ***value2***, ***value1*** >>>
After: …, ***value1***, ***value2***, ***value1***

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

This instruction must not be used if ***value1*** or ***value2*** is of type **double** or **long**.

**Mnemonic:** **dup_x2**

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

**Opcode:** 91 (0x5B)

**Operands:** None

**Operand Stack:** Before: …, ***value3***, ***value2***, ***value1*** >>>
After: …, ***value1***, ***value3***, ***value2***, ***value1***

***value1***, ***value2***, and ***value3*** must not be of type **double** or **long**.

Before: …, ***value2***, ***value1*** >>>
After: …, ***value1***, ***value2***, ***value1***

***value1*** must not be of type **double** or **long**. ***value2*** must be of type **double** or **long**.

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

**Mnemonic:** **dup2**

**Operation:** Duplicate the top one or two operand stack values.

**Opcode:** 92 (0x5C)

**Operands:** None

**Operand Stack:** Before: …, ***value2***, ***value1*** >>>
After: …, ***value2***, ***value1***, ***value2***, ***value1***

***value1*** and ***value2*** must not be of type **double** or **long**.

Before: …, ***value*** >>>
After: …, ***value***, ***value***

***value*** must be of type **double** or **long**.

**Description:** Duplicate the top one or two values on the operand stack and push the duplicated value(s) back to that stack in the original order.

**Mnemonic:** **dup2_x1**

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

**Opcode:** 93 (0x5D)

**Operands:** None

**Operand Stack:** Before: …, ***value3***, ***value2***, ***value1*** >>>
After: …, ***value2***, ***value1***, ***value3***, ***value2***, ***value1***

***value1***, ***value2***, and ***value3*** must not be of type **double** or **long**.

Before: …, ***value2***, ***value1*** >>>
After: …, ***value1***, ***value2***, ***value1***

***value1*** must be of type **double** or **long**. ***value2*** must not be of type **double** or **long**.

**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(s) in that stack.

**Mnemonic:** **dup2_x2**

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

**Opcode:** 94 (0x5E)

**Operands:** None

**Operand Stack:** Before: …, ***value4***, ***value3***, ***value2***, ***value1*** >>>
After: …, ***value2***, ***value1***, ***value4***, ***value3***, ***value2***, ***value1***

***value1***, ***value2***, ***value3***, and ***value4*** must not be of type **double** or **long**.

Before: …, ***value3***, ***value2***, ***value1*** >>>
After: …, ***value1***, ***value3***, ***value2***, ***value1***

***value1*** must be of type **double** or **long**. ***value2*** and ***value3*** must not be of type **double** or **long**.

Before: …, ***value3***, ***value2***, ***value1*** >>>
After: …, ***value2***, ***value1***, ***value3***, ***value2***, ***value1***

***value1*** and ***value2*** must not be of type **double** or **long**. ***value3*** must be of type **double** or **long**.

Before: …, ***value2***, ***value1*** >>>
After: …, ***value1***, ***value2***, ***value1***

***value1*** and ***value2*** must be of type **double** or **long**.

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

**Mnemonic:** **f2d**

**Operation:** Convert **float** to **double**.

**Opcode:** 141 (0x8D)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** The ***value*** on the top of the operand stack must be of **float** type. It pops from that stack and undergoes value set conversion, resulting in ***value'***. Then ***value'*** converts to a **double** ***result***, which pushes to the operand stack.

**Notes:** Where an **f2d** instruction is FP-strict, it performs a widening primitive conversion. The conversion is exact because all values of the float value set are exactly representable by values of the double value set.

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 ***value*** is taken from the float-extended-exponent value set and ***result*** is constrained to the double value set, rounding of ***value*** may be required.

**Mnemonic:** **f2i**

**Operation:** Convert **float** to **int**.

**Opcode:** 139 (0x8B)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** The ***value*** on the top of the operand stack must be of **float** type. It pops from that stack and undergoes value set conversion, resulting in ***value'***. Then ***value'*** converts to an **int** ***result***. The ***result*** pushes to the operand stack:

1) If the ***value'*** is NaN, the ***result*** of the conversion is an **int** 0.

2) Otherwise, if the ***value'*** is not an infinity, it rounds to an integer value ***v***, rounding towards zero using IEEE 754 round towards zero mode. If this integer value ***v*** can represent as an **int**, the ***result*** is the **int** value ***v***.

3) 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:** This instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of ***value'*** and may also lose precision.

**Mnemonic:** **f2l**

**Operation:** Convert **float** to **long**.

**Opcode:** 140 (0x8C)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** The ***value*** on the top of the operand stack must be of **float** type. It pops from that stack and undergoes value set conversion, resulting in ***value'***. Then ***value'*** converts to a **long** ***result***. The ***result*** pushes to the operand stack:

1) If the ***value'*** is NaN, the ***result*** of the conversion is a **long** 0.

2) Otherwise, if the ***value'*** is not an infinity, it rounds to an integer value ***v***, rounding towards zero using IEEE 754 round towards zero mode. If this integer value ***v*** can represent as a **long**, the ***result*** is the **long** value ***v***.

3) 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:** This instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of ***value'*** and may also lose precision.

**Mnemonic:** **fadd**

**Operation:** Add **float**.

**Opcode:** 98 (0x62)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **float** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **float** ***result*** is ***value1'***+***value2'***, which pushes to the operand stack.

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

1) If either ***value1'*** or ***value2'*** is NaN, the result is NaN.

2) The sum of two infinities of opposite sign is NaN.

3) The sum of two infinities of the same sign is the infinity of that sign.

4) The sum of an infinity and any finite value is equal to the infinity.

5) The sum of two zeroes of opposite sign is positive zero.

6) The sum of two zeroes of the same sign is the zero of that sign.

7) The sum of a zero and a nonzero finite value is equal to the nonzero value.

8) The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.

9) In the remaining cases, where neither ***value1*** nor ***value2*** is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum computes and rounds to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a **float**, the operation is said to overflow; the result is an infinity of appropriate sign. If the magnitude is too small to represent as a **float**, the operation is said to underflow; the result is a zero of appropriate sign.

The JVM requires support of gradual underflow, as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of this instruction never throws a runtime exception.

| | |
|---|---|
| **Mnemonic:** | **faload** |
| **Operation:** | Load **float** from array. |
| **Opcode:** | 48 (0x30) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref***, ***index*** >>><br>After: …, ***value*** |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array whose components are of **float** type. Furthermore, ***index*** must be of **int** type. Both ***arrayref*** and ***index*** pop from the operand stack. The **float** ***value*** in the array component at ***index*** retrieves and pushes to the operand stack. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***arrayref*** is **null**.<br><br>This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array. |

| | |
|---|---|
| **Mnemonic:** | **fastore** |
| **Operation:** | Store into **float** array. |
| **Opcode:** | 81 (0x51) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref***, ***index***, ***value*** >>><br>After: … |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array whose components are of **float** type. Furthermore, ***index*** must be of **int** type and ***value*** must be of **float** type. ***arrayref***, ***index***, and ***value*** pop from the operand stack. The **float *value*** undergoes value set conversion, resulting in ***value'***, which stores as the array component at ***index***. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***arrayref*** is **null**.<br><br>This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array. |

**Mnemonic:** **fcmpg**

**Operation:** Compare **float**s: which is greater.

**Opcode:** 150 (0x96)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **float** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. A floating-point comparison is then performed:

1) If ***value1'*** is greater than ***value2'***, **int** value 1 pushes to the operand stack.

2) If ***value1'*** is equal to ***value2'***, **int** value 0 pushes to the operand stack.

3) If ***value1'*** is less than ***value2'***, **int** value -1 pushes to the operand stack.

4) If at least one of ***value1'*** or ***value2'*** is NaN, **int** value 1 pushes to the operand stack.

IEEE 754 rules for floating-point comparison are observed. All values other than NaN are ordered, with negative infinity being less than all finite values, and positive infinity being greater than all finite values. Positive zero and negative zero are considered to be equal.

**Notes:** **fcmpg** and **fcmpl** differ only in their treatment of a comparison involving NaN.

**Mnemonic:** **fcmpl**

**Operation:** Compare **float**s: which is lesser.

**Opcode:** 149 (0x95)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **float** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. A floating-point comparison is then performed:

1) If ***value1'*** is greater than ***value2'***, **int** value 1 pushes to the operand stack.

2) If ***value1'*** is equal to ***value2'***, **int** value 0 pushes to the operand stack.

3) If ***value1'*** is less than ***value2'***, **int** value -1 pushes to the operand stack.

4) If at least one of ***value1'*** or ***value2'*** is NaN, **int** value -1 pushes to the operand stack.

IEEE 754 rules for floating-point comparison are observed. All values other than NaN are ordered, with negative infinity being less than all finite values, and positive infinity being greater than all finite values. Positive zero and negative zero are considered to be equal.

**Notes:** **fcmpg** and **fcmpl** differ only in their treatment of a comparison involving NaN.

**Mnemonic:** **fconst_0**

**Operation:** Push **float 0.0f**.

**Opcode:** 11 (0x0B)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **0.0**

**Description:** Push the **float** constant 0.0 to the operand stack.

**Mnemonic:** **fconst_1**

**Operation:** Push **float 1.0f**.

**Opcode:** 12 (0x0C)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **1.0**

**Description:** Push the **float** constant 1.0 to the operand stack.

**Mnemonic:** **fconst_2**

**Operation:** Push **float 2.0f**.

**Opcode:** 13 (0x0D)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **2.0**

**Description:** Push the **float** constant 2.0 to the operand stack.

**Mnemonic:** **fdiv**

**Operation:** Divide **float**.

**Opcode:** 110 (0x6E)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **float** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **float** ***result*** from ***value1'***/***value2'*** pushes to the operand stack.

The following IEEE arithmetic rules govern this instruction:

1) If either ***value1'*** or ***value2'*** is NaN, the result is NaN.

2) If neither ***value1'*** nor ***value2'*** is NaN, the result's sign is positive (if both values have the same sign) or negative (if the values have different signs).

3) Division of positive/negative infinity by positive/negative infinity yields NaN.

4) Division of positive/negative infinity by a finite value results in a signed infinity (where the sign is based on Rule 2).

5) Division of a finite value by positive/negative infinity results in a signed zero (where the sign is based on Rule 2).

6) Division of positive/negative zero by positive/negative zero yields NaN. Division of zero by any other finite value results in a signed zero (where the sign is based on Rule 2).

7) Division of a nonzero finite value by positive/negative zero results in a signed infinity (where the sign is based on Rule 2).

8) In remaining cases, where neither ***value1*** nor ***value2*** is an infinity, a zero, or NaN, the quotient computes and rounds to the nearest **float** using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a **float**, the operation is said to overflow and the result is an infinity of the appropriate sign. If the magnitude is too small to represent as a **float**, the operation is said to underflow and the result is a zero of the appropriate sign.

The JVM requires support of gradual underflow as defined by IEEE 754. This instruction never throws an exception, even though overflow, underflow, division by zero, or loss of precision may occur.

**Mnemonic:** **fload**

**Operation:** Load **float** from local variable.

**Opcode:** 23 (0x17)

**Operands:** ***index***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** ***index*** is an unsigned byte that must be an index into the current stack frame's local variable array. Furthermore, the local variable at ***index*** must contain a **float**. The ***value*** of the local variable at ***index*** pushes to the operand stack.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **fload_0**

**Operation:** Load **float** from local variable 0.

**Opcode:** 34 (0x22)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 0 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 0 must contain a **float**. The ***value*** of the local variable at 0 pushes to the operand stack.

**Notes:** This instruction is the same as **fload** (with a 0 index operand), except that the 0 in **fload_0** is implicit.

**Mnemonic:** **fload_1**

**Operation:** Load **float** from local variable 1.

**Opcode:** 35 (0x23)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 1 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 1 must contain a **float**. The ***value*** of the local variable at 1 pushes to the operand stack.

**Notes:** This instruction is the same as **fload** (with a 1 index operand), except that the 1 in **fload_1** is implicit.

**Mnemonic:** **fload_2**

**Operation:** Load **float** from local variable 2.

**Opcode:** 36 (0x24)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 2 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 2 must contain a **float**. The ***value*** of the local variable at 2 pushes to the operand stack.

**Notes:** This instruction is the same as **fload** (with a 2 index operand), except that the 2 in **fload_2** is implicit.

**Mnemonic:** **fload_3**

**Operation:** Load **float** from local variable 3.

**Opcode:** 37 (0x25)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 3 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 3 must contain a **float**. The ***value*** of the local variable at 3 pushes to the operand stack.

**Notes:** This instruction is the same as **fload** (with a 3 index operand), except that the 3 in **fload_3** is implicit.

**Mnemonic:** **fmul**

**Operation:** Multiply **float**.

**Opcode:** 106 (0x6A)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **float** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **float** ***result*** from ***value1'*******value2'*** pushes to the operand stack.

The following IEEE arithmetic rules govern this instruction:

1) If either ***value1'*** or ***value2'*** is NaN, the result is NaN.

2) If neither ***value1'*** nor ***value2'*** is NaN, the result's sign is positive (if both values have the same sign) or negative (if the values have different signs).

3) Multiplication of positive/negative infinity by positive/negative zero yields NaN.

4) Multiplication of positive/negative infinity by a finite value results in a signed infinity (where the sign is based on Rule 2).

5) In remaining cases, where neither an infinity nor NaN is involved, the product computes and rounds to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a **float**, the operation is said to overflow and the result is an infinity of the appropriate sign. If the magnitude is too small to represent as a **float**, the operation is said to underflow and the result is a zero of the appropriate sign.

The JVM requires support of gradual underflow as defined by IEEE 754. This instruction never throws an exception, even though overflow, underflow, or loss of precision may occur.

**Mnemonic:** **fneg**

**Operation:** Negate **float**.

**Opcode:** 118 (0x76)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value*** must be of **float** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value1'***. The **float** ***result*** is the arithmetic negation of ***value1***, which pushes to the operand stack.

Negation is not the same as subtraction from zero, for **float** values. If ***x*** is +0.0 then 0.0-***x*** equals +0.0, but –***x*** equals –0.0. Unary minus inverts the sign of a **float**.

There are three special cases:

1) If ***value*** is NaN, the result is NaN. (NaN has no sign.)

2) If ***value*** is positive/negative infinity, the result is the infinity of the opposite sign.

3) If ***value*** is positive/negative zero, the result is the zero of opposite sign.

**Mnemonic:** **frem**

**Operation:** Remainder **float**.

**Opcode:** 114 (0x72)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **float** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **float** ***result*** calculates and pushes to the operand stack.

This instruction behaves in a manner similar to that of the **irem** and **lrem** JVM instructions.

The following rules govern this instruction:

1) If either ***value1'*** or ***value2'*** is NaN, the result is NaN.

2) If neither ***value1'*** nor ***value2'*** is NaN, the result's sign equals the dividend's sign.

3) If the dividend is positive/negative infinity or the divisor is positive/negative zero (or both), the result is NaN.

4) If the dividend is finite and the divisor is positive/negative infinity, the result equals the dividend.

5) If the dividend is positive/negative zero and the divisor is finite, the result equals the dividend.

6) In remaining cases, where neither ***value1*** nor ***value2*** is a positive/negative infinity, a positive/negative zero, nor NaN, the floating-point remainder ***result*** from a dividend ***value1'*** and a divisor ***value2'*** is defined by the mathematical relation ***result***=***value1'***–(***value2'*******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'***.

Even though division by zero may occur, this instruction never throws a runtime exception. Overflow, underflow, or precision loss cannot occur.

**Notes:** This instruction's result is not the same as that of the IEEE 754 remainder operation. Use the **Math.IEEEremainder** library routine to compute that operation.

**Mnemonic:** **freturn**

**Operation:** Return **float** from method.

**Opcode:** 174 (0xAE)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: [empty]

**Description:** Both the current method's return type and ***value***'s type must be **float**. If the current method is **synchronized**, the monitor acquired or reentered on that method's invocation is released or exited (respectively) as if by executing the **monitorexit** instruction. If no exception is thrown, ***value*** pops from the current stack frame's operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** pushes to the operand stack of the invoker's stack frame. Any other values on the current method's operand stack discard.

The interpreter returns control to the current method's invoker, reinstating the invoker's stack frame as the current stack frame.

**Runtime Exceptions:** This instruction throws an **IllegalMonitorStateException** 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.

This instruction also throws an **IllegalMonitorStateException** if the JVM implementation enforces the rules on structured lock usage, and the number of lock operations performed by a thread on a lock does not match the number of unlock operations by that thread on that lock (whether the method invocation completes normally or abruptly).

**Mnemonic:** **fstore**

**Operation:** Store **float** into local variable.

**Opcode:** 56 (0x38)

**Operands:** ***index***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***index*** is an unsigned byte that must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **float** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variable at ***index***.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **fstore_0**

**Operation:** Store **float** into local variable 0.

**Opcode:** 67 (0x43)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 0 must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **float** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variable at 0.

**Notes:** This instruction is the same as **fstore** (with a 0 index operand), except that the 0 in **fstore_0** is implicit.

**Mnemonic:** **fstore_1**

**Operation:** Store **float** into local variable 1.

**Opcode:** 68 (0x44)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 1 must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **float** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variable at 1.

**Notes:** This instruction is the same as **fstore** (with a 1 index operand), except that the 1 in **fstore_1** is implicit.

**Mnemonic:** **fstore_2**

**Operation:** Store **float** into local variable 2.

**Opcode:** 69 (0x45)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 2 must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **float** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variable at 2.

**Notes:** This instruction is the same as **fstore** (with a 2 index operand), except that the 2 in **fstore_2** is implicit.

**Mnemonic:** **fstore_3**

**Operation:** Store **float** into local variable 3.

**Opcode:** 70 (0x46)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 3 must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **float** type. It pops from the operand stack and undergoes value set conversion, resulting in ***value'***. ***value'*** stores in the local variable at 3.

**Notes:** This instruction is the same as **fstore** (with a 3 index operand), except that the 3 in **fstore_3** is implicit.

**Mnemonic:** **fsub**

**Operation:** Subtract **float**.

**Opcode:** 102 (0x66)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **float** type. Those values pop from the operand stack and undergo value set conversion, resulting in ***value1'*** and ***value2'***. The **float** ***result*** is ***value1'-value2'***, which pushes to the operand stack.

For **float** subtraction, it is always the case that ***a-b*** results in the same value as ***a*** +(-***b***). However, subtraction from zero is not the same as negation: if ***x*** is +0.0, 0.0-***x*** equals +0.0, but -***x*** equals -0.0.

The JVM requires support of gradual underflow, as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of this instruction never throws a runtime exception.

| | |
|---|---|
| **Mnemonic:** | **getfield** |
| **Operation:** | Get field from object. |
| **Opcode:** | 180 (0xB4) |
| **Operands:** | ***indexbyte1***, ***indexbyte2*** |
| **Operand Stack:** | Before: …, ***objectref*** >>><br>After: …, ***value*** |
| **Description:** | ***objectref***, which must be of **reference** type, pops from the operand stack. The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index value is (***indexbyte1***<<8)\|***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to a field, 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 resolves. The ***value*** of the referenced field in ***objectref*** fetches and pushes to the operand stack.<br><br>***objectref***'s class must not be an array. If the field is **protected**, and is either a member of the current class or a member of a superclass of the current class, ***objectref***'s class 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 of the JVM specification can be thrown.<br><br>This instruction throws an **IncompatibleClassChangeError** if the resolved field is a **static** field. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***objectref*** is **null**. |
| **Notes:** | This instruction cannot be used to access an array's **length** field. Use **arraylength** instead. |

| | |
|---|---|
| **Mnemonic:** | **getstatic** |
| **Operation:** | Get **static** field from class. |
| **Opcode:** | 178 (0xB2) |
| **Operands:** | ***indexbyte1***, ***indexbyte2*** |
| **Operand Stack:** | Before: … >>><br>After: …, ***value*** |
| **Description:** | The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index value is (***indexbyte1***<<8)\|***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to a field, 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 resolves.<br><br>On successful resolution of the field, the class or interface that declares the resolved field initializes (unless already initialized).<br><br>The ***value*** of the class or interface field fetches and pushes to 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 of the JVM specification can be thrown.<br><br>This instruction throws an **IncompatibleClassChangeError** if the resolved field is not a class (**static**) field or an interface field. |
| **Runtime Exceptions:** | If execution of this instruction causes initialization of the referenced class or interface, the instruction may throw an **Error** as detailed in section 2.17.5 of the JVM specification. |

**Mnemonic:** **goto**

**Operation:** Branch always.

**Opcode:** 167 (0xA7)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** No change

**Description:** Unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **goto** instruction.

**Mnemonic:** **goto_w**

**Operation:** Branch always (wide index).

**Opcode:** 200 (0xC8)

**Operands:** ***branchbyte1***, ***branchbyte2***, ***branchbyte3***, ***branchbyte4***

**Operand Stack:** No change

**Description:** Unsigned bytes ***branchbyte1***, ***branchbyte2***, ***branchbyte3***, and ***branchbyte4*** construct into a 32-bit ***branchoffset***, via (***branchbyte1***<<24)|(***branchbyte2***<<16)|(***branchbyte3***<<8)|***branchbyte4***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **goto_w** instruction.

**Notes:** Despite the 32-bit branch address, other factors may limit the size of a method to 65535 bytes. Future releases of the JVM may raise this limit.

**Mnemonic:** **i2b**

**Operation:** Convert **int** to **byte**.

**Opcode:** 145 (0x91)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **int** type. It pops from the operand stack, truncates to a **byte**, and then sign-extends to an **int** ***result*** - which pushes to the operand stack.

**Notes:** This instruction performs a narrowing primitive conversion. As a result, information about ***value***'s overall magnitude may be lost. Furthermore, ***result*** and ***value*** may have different signs.

**Mnemonic:** **i2c**

**Operation:** Convert **int** to **char**.

**Opcode:** 146 (0x92)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **int** type. It pops from the operand stack, truncates to a **char**, and then zero-extends to an **int** ***result*** - which pushes to the operand stack.

**Notes:** This instruction performs a narrowing primitive conversion. As a result, information about ***value***'s overall magnitude may be lost. Furthermore, ***result*** (which is always positive) and ***value*** may have different signs.

**Mnemonic:** **i2d**

**Operation:** Convert **int** to **double**.

**Opcode:** 135 (0x87)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **int** type. It pops from the operand stack and converts to a **double** ***result*** - which pushes to the operand stack.

**Notes:** This instruction performs a widening primitive conversion. Because all **int** values are exactly representable by **double** type, the conversion is exact.

**Mnemonic:** **i2f**

**Operation:** Convert **int** to **float**.

**Opcode:** 134 (0x86)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **int** type. It pops from the operand stack and converts to a **float** ***result*** (using IEEE 754 round to nearest mode) - which pushes to the operand stack.

**Notes:** This instruction performs a widening primitive conversion, but may result in a loss of precision because **float** values have only 24 significand bits.

**Mnemonic:** **i2l**

**Operation:** Convert **int** to **long**.

**Opcode:** 133 (0x85)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **int** type. It pops from the operand stack and sign-extends to a **long** ***result*** - which pushes to the operand stack.

**Notes:** This instruction performs a widening primitive conversion. Because all **int** values are exactly representable by type **long**, the conversion is exact.

**Mnemonic:** **i2s**

**Operation:** Convert **int** to **short**.

**Opcode:** 147 (0x93)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **int** type. It pops from the operand stack, truncates to a **short**, and then sign-extends to an **int** ***result*** - which pushes to the operand stack.

**Notes:** This instruction performs a narrowing primitive conversion. As a result, information about ***value***'s overall magnitude may be lost. Furthermore, ***result*** and ***value*** may have different signs.

**Mnemonic:** **iadd**

**Operation:** Add **int**.

**Opcode:** 96 (0x60)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. The **int** ***result*** from ***value1***+***value2*** pushes to 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 **int** type. If overflow occurs, the result's sign may not be the same as the sign of the mathematical sum of both values.

This instruction never throws an exception, even though overflow may occur.

| | |
|---|---|
| **Mnemonic:** | **iaload** |
| **Operation:** | Load **int** from array. |
| **Opcode:** | 46 (0x2E) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref***, ***index*** >>><br>After: …, ***value*** |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array whose components are of **int** type. Furthermore, ***index*** must be of **int** type. Both ***arrayref*** and ***index*** pop from the operand stack. The **int** ***value*** in the array component at ***index*** retrieves and pushes to the operand stack. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***arrayref*** is **null**.<br><br>This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array. |

**Mnemonic:** **iand**

**Operation:** Boolean AND **int**.

**Opcode:** 126 (0x7E)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. The **int** ***result*** from a bitwise AND (conjunction) of both values pushes to the operand stack.

| | |
|---|---|
| **Mnemonic:** | **iastore** |
| **Operation:** | Store into **int** array. |
| **Opcode:** | 79 (0x4F) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref***, ***index***, ***value*** >>><br>After: … |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array whose components are of **int** type. Furthermore, ***index*** and ***value*** must both be of **int** type. ***arrayref***, ***index***, and ***value*** pop from the operand stack. The **int** ***value*** stores as the array component at ***index***. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***arrayref*** is **null**.<br><br>This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array. |

**Mnemonic:** **iconst_0**

**Operation:** Push **int** constant **0**.

**Opcode:** 3 (0x03)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **0**

**Description:** Push the **int** constant 0 to the operand stack.

**Notes:** This instruction is equivalent to **bipush 0**, except that 0 is implicit in **iconst_0**.

**Mnemonic:** **iconst_1**

**Operation:** Push **int** constant **1**.

**Opcode:** 4 (0x04)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **1**

**Description:** Push the **int** constant 1 to the operand stack.

**Notes:** This instruction is equivalent to **bipush 1**, except that 1 is implicit in **iconst_1**.

**Mnemonic:** **iconst_2**

**Operation:** Push **int** constant **2**.

**Opcode:** 5 (0x05)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **2**

**Description:** Push the **int** constant 2 to the operand stack.

**Notes:** This instruction is equivalent to **bipush 2**, except that 2 is implicit in **iconst_2**.

**Mnemonic:** **iconst_3**

**Operation:** Push **int** constant **3**.

**Opcode:** 6 (0x06)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **3**

**Description:** Push the **int** constant 3 to the operand stack.

**Notes:** This instruction is equivalent to **bipush 3**, except that 3 is implicit in **iconst_3**.

**Mnemonic:** **iconst_4**

**Operation:** Push **int** constant **4**.

**Opcode:** 7 (0x07)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **4**

**Description:** Push the **int** constant 4 to the operand stack.

**Notes:** This instruction is equivalent to **bipush 4**, except that 4 is implicit in **iconst_4**.

**Mnemonic:** **iconst_5**

**Operation:** Push **int** constant **5**.

**Opcode:** 8 (0x08)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **5**

**Description:** Push the **int** constant 5 to the operand stack.

**Notes:** This instruction is equivalent to **bipush 5**, except that 5 is implicit in **iconst_5**.

**Mnemonic:** **iconst_m1**

**Operation:** Push **int** constant **-1**.

**Opcode:** 2 (0x02)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **-1**

**Description:** Push the **int** constant -1 to the operand stack.

**Notes:** This instruction is equivalent to **bipush -1**, except that -1 is implicit in **iconst_m1**.

**Mnemonic:** **idiv**

**Operation:** Divide **int**.

**Opcode:** 108 (0x6C)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. The **int** ***result*** from ***value1***/***value2*** pushes to the operand stack.

This division rounds towards zero: the quotient produced for **int** values in ***n***/***d*** is an **int** value ***q*** whose magnitude is as large as possible while satisfying |***d***.***q***|<=|***n***|. ***q*** is positive when |***n***|>=|***d***| and ***n*** and ***d*** have the same sign. ***q*** is negative when |***n***|>=|***d***| and ***n*** and ***d*** have opposite signs.

One special case 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, overflow occurs and the result equals the dividend. No exception is thrown in this case.

**Runtime Exceptions:** This instruction throws an **ArithmeticException** if the divisor's value is 0.

**Mnemonic:** **if_acmpeq**

**Operation:** Branch if **reference**s are equal.

**Opcode:** 165 (0xA5)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …

**Description:** ***value1*** and ***value2*** must be of **reference** type. They pop from the operand stack and compare. If both are equal, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **if_acmpeq** instruction.

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

**Mnemonic:** **if_acmpne**

**Operation:** Branch if **reference**s are not equal.

**Opcode:** 166 (0xA6)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …

**Description:** ***value1*** and ***value2*** must be of **reference** type. They pop from the operand stack and compare. If both are not equal, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **if_acmpne** instruction.

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

**Mnemonic:** **if_icmpeq**

**Operation:** Branch if **int**s are equal.

**Opcode:** 159 (0x9F)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …

**Description:** ***value1*** and ***value2*** must be of **int** type. They pop from the operand stack and compare. If both are equal, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **if_icmpeq** instruction.

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

**Mnemonic:** **if_icmpge**

**Operation:** Branch if first **int** greater than or equal to second **int**.

**Opcode:** 162 (0xA2)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …

**Description:** ***value1*** and ***value2*** must be of **int** type. They pop from the operand stack and compare. If ***value1*** is greater than or equal to ***value2***, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **if_icmpge** instruction.

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

**Mnemonic:** **if_icmpgt**

**Operation:** Branch if first **int** greater than second **int**.

**Opcode:** 163 (0xA3)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …

**Description:** ***value1*** and ***value2*** must be of **int** type. They pop from the operand stack and compare. If ***value1*** is greater than ***value2***, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **if_icmpgt** instruction.

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

**Mnemonic:** **if_icmple**

**Operation:** Branch if first **int** less than or equal to second **int**.

**Opcode:** 164 (0xA4)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …

**Description:** ***value1*** and ***value2*** must be of **int** type. They pop from the operand stack and compare. If ***value1*** is less than or equal to ***value2***, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **if_icmple** instruction.

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

**Mnemonic:** **if_icmplt**

**Operation:** Branch if first **int** less than second **int**.

**Opcode:** 161 (0xA1)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …

**Description:** ***value1*** and ***value2*** must be of **int** type. They pop from the operand stack and compare. If ***value1*** is less than ***value2***, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **if_icmplt** instruction.

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

**Mnemonic:** **if_icmpne**

**Operation:** Branch if **int**s are not equal.

**Opcode:** 160 (0xA0)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …

**Description:** ***value1*** and ***value2*** must be of **int** type. They pop from the operand stack and compare. If both are not equal, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **if_icmpne** instruction.

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

**Mnemonic:** **ifeq**

**Operation:** Branch if **int** equals zero.

**Opcode:** 153 (0x99)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***value*** must be of **int** type. It pops from the operand stack and compares against zero. If ***value*** equals zero, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **ifeq** instruction.

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

**Mnemonic:** **ifge**

**Operation:** Branch if **int** greater than or equal to zero.

**Opcode:** 156 (0x9C)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***value*** must be of **int** type. It pops from the operand stack and compares against zero. If ***value*** is greater than or equal to zero, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **ifge** instruction.

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

**Mnemonic:** **ifgt**

**Operation:** Branch if **int** greater than zero.

**Opcode:** 157 (0x9D)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***value*** must be of **int** type. It pops from the operand stack and compares against zero. If ***value*** is greater than zero, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **ifgt** instruction.

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

**Mnemonic:** **ifle**

**Operation:** Branch if **int** less than or equal to zero.

**Opcode:** 158 (0x9E)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***value*** must be of **int** type. It pops from the operand stack and compares against zero. If ***value*** is less than or equal to zero, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **ifle** instruction.

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

**Mnemonic:** **iflt**

**Operation:** Branch if **int** less than zero.

**Opcode:** 155 (0x9B)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***value*** must be of **int** type. It pops from the operand stack and compares against zero. If ***value*** is less than zero, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **iflt** instruction.

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

**Mnemonic:** **ifne**

**Operation:** Branch if **int** not equal to zero.

**Opcode:** 154 (0x9A)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***value*** must be of **int** type. It pops from the operand stack and compares against zero. If ***value*** is not equal to zero, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **ifne** instruction.

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

**Mnemonic:** **ifnonnull**

**Operation:** Branch if **reference** not **null**.

**Opcode:** 199 (0xC7)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***value*** must be of **reference** type. It pops from the operand stack. If ***value*** is not **null**, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **ifnonnull** instruction.

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

**Mnemonic:** **ifnull**

**Operation:** Branch if **reference** is **null**.

**Opcode:** 198 (0xC6)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***value*** must be of **reference** type. It pops from the operand stack. If ***value*** is **null**, unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction's opcode. The target address must be that of an instruction opcode within the method that contains the **ifnull** instruction.

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

**Mnemonic:** **iinc**

**Operation:** Increment local variable by constant.

**Opcode:** 132 (0x84)

**Operands:** ***index***, ***const***

**Operand Stack:** No change

**Description:** ***index*** is an unsigned byte that must be an index into the current stack frame's local variable array. Furthermore, the local variable at ***index*** must contain an **int**. ***const*** is an immediate signed byte. ***const*** first sign-extends to an **int**, and then the local variable at ***index*** increments by that amount.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index and to increment it by a two-byte immediate value.

**Mnemonic:** **iload**

**Operation:** Load **int** from local variable.

**Opcode:** 21 (0x15)

**Operands:** ***index***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** ***index*** is an unsigned byte that must be an index into the current stack frame's local variable array. Furthermore, the local variable at ***index*** must contain an **int**. The ***value*** of the local variable at ***index*** pushes to the operand stack.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **iload_0**

**Operation:** Load **int** from local variable 0.

**Opcode:** 26 (0x1A)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 0 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 0 must contain an **int**. The ***value*** of the local variable at 0 pushes to the operand stack.

**Notes:** This instruction is the same as **iload** (with a 0 index operand), except that the 0 in **iload_0** is implicit.

**Mnemonic:** **iload_1**

**Operation:** Load **int** from local variable 1.

**Opcode:** 27 (0x1B)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 1 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 1 must contain an **int**. The ***value*** of the local variable at 1 pushes to the operand stack.

**Notes:** This instruction is the same as **iload** (with a 1 index operand), except that the 1 in **iload_1** is implicit.

**Mnemonic:** **iload_2**

**Operation:** Load **int** from local variable 2.

**Opcode:** 28 (0x1C)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 2 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 2 must contain an **int**. The ***value*** of the local variable at 2 pushes to the operand stack.

**Notes:** This instruction is the same as **iload** (with a 2 index operand), except that the 2 in **iload_2** is implicit.

**Mnemonic:** **iload_3**

**Operation:** Load **int** from local variable 3.

**Opcode:** 29 (0x1D)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 3 must be an index into the current stack frame's local variable array. Furthermore, the local variable at 3 must contain an **int**. The ***value*** of the local variable at 3 pushes to the operand stack.

**Notes:** This instruction is the same as **iload** (with a 3 index operand), except that the 3 in **iload_3** is implicit.

**Mnemonic:** **imul**

**Operation:** Multiply **int**.

**Opcode:** 104 (0x68)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. The **int** ***result*** from ***value1*******value2*** pushes to the operand stack.

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

This instruction never throws an exception, even though overflow may occur.

**Mnemonic:** **ineg**

**Operation:** Negate **int**.

**Opcode:** 116 (0x74)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value*** must be of **int** type. It pops from the operand stack. The **int** ***result*** is the arithmetic negation of ***value***, -***value***, which pushes to the operand stack.

Negation is the same as subtraction from zero, for **int** values. Because the JVM uses two's-complement representation for integers and the range of two's-complement values is asymmetric, the negation of the maximum negative integer results in the same maximum negative integer. Despite this overflow, no exception is thrown.

For all **int** values ***x***, -***x*** equals (~***x***)+1.

**Mnemonic:** **instanceof**

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

**Opcode:** 193 (0xC1)

**Operands:** ***indexbyte1***, ***indexbyte2***

**Operand Stack:** Before: …, ***objectref*** >>>
After: …, ***result***

**Description:** ***objectref***, which must be of **reference** type, pops from the operand stack. The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, via (***indexbyte1***<<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 then resolves.

If ***objectref*** is not **null** and is an instance of the resolved class or array or implements the resolved interface, this instruction pushes an **int** ***result*** of 1 to the operand stack. Otherwise, an **int** ***result*** of 0 pushes.

The following rules determine whether a non-**null** ***objectref*** is an instance of the resolved type: If *S* is the class of the object referred to be ***objectref*** and *T* is the resolved class, array, or interface type, this instruction determines whether ***objectref*** is an instance of type *T* as follows:

1) If *S* is an ordinary (nonarray) class, then:

1.1) If *T* is a class type, then *S* must be the same class as *T* or a subclass of *T*.

1.2) If *T* is an interface type, then *S* must implement interface *T*.

2) If *S* is an interface type, then:

2.1) If *T* is a class type, then *T* must be **Object**.

2.2) If *T* is an interface type, then *T* must be the same interface as *S* or a superinterface of *S*.

3) If *S* is a class representing the array type *SC*[] (an array of components of type *SC*), then:

3.1) If *T* is a class type, then *T* must be **Object**.

3.2) If *T* is an array type *TC*[] (an array of components of type *TC*), then one of the following must be true:

3.2.1) *TC* and *SC* are the same primitive type.

3.2.2) *TC* and *SC* are reference types, and type *SC* can be cast to *TC* by recursive

application of these rules.

3.3) If *T* is an interface type, *T* must be one of the interfaces implemented by arrays.

**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 of the JVM specification can be thrown.

**Notes:** This instruction is very similar to **checkcast**. However, **instanceof** differs in its treatment of **null**, its behavior when its test fails (**checkcast** throws an exception, whereas **instanceof** pushes a result code), and its effect on the operand stack.

**Mnemonic:** **invokeinterface**

**Operation:** Invoke interface method.

**Opcode:** 185 (0xB9)

**Operands:** ***indexbyte1***, ***indexbyte2***, ***count***, ***0***

**Operand Stack:** Before: …, ***objectref***, [***arg1***, [***arg2*** …]] >>>
After: …

**Description:** The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index's value is (***indexbyte1***<<8)| ***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to an interface method, which gives the name and descriptor 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 resolves. The interface method must not be an instance initialization method or the class or interface initialization method.

***count*** is an unsigned byte that must not be zero. Following ***count*** is a single 0 byte.

***objectref*** must be of **reference** type 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 resolved interface method's descriptor.

Let *C* be ***objectref***'s class. The actual method to be invoked is selected by the following lookup procedure:

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

2) 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.

3) Otherwise, an **AbstractMethodError** is raised.

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

If the method is not **native**, the ***nargs*** argument values and ***objectref*** pop from the operand stack. A new stack frame creates on the JVM stack for the method being invoked. The ***objectref*** and argument values are consecutively made the values of local variables of the new stack frame, with ***objectref*** in local variable 0, ***arg1*** in local variable 1 (or, if ***arg1*** is of **long** or **double** type, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion prior to being stored in a local variable. The new stack frame is then made current, and the JVM **pc** register 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 into the JVM, that bounding is done. The ***nargs*** argument values and ***objectref*** pop from the operand stack and pass as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion prior to being passed as a parameter. The parameters pass and the code invokes in an implementation-dependent manner. When the platform-dependent code returns:

1) If the **native** method is synchronized, the monitor associated with ***objectref*** releases or exits as if by execution of a **monitorexit** instruction.

2) If the **native** method returns a value, the return value of the platform-dependent code converts in an implementation-dependent way to the return type of the **native** method and pushes to the operand stack.

**Linking Exceptions:** During resolution of the symbolic reference to the interface method, any of the exceptions documented in section 5.4.3.4 of the JVM specification can be thrown.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***objectref*** is **null**.

This instruction throws an **IncompatibleClassChangeError** if ***objectref*** 's class does not implement the resolved interface.

This instruction throws an **AbstractMethodError** if no method matching the resolved name and descriptor selects, or if the selected method is **abstract**.

This instruction throws an **IllegalAccessError** if the selected method is not **public**.

This instruction throws an **UnsatisfiedLinkError** if the selected method is **native** and the code that implements the method cannot be bound.

**Notes:** ***count*** records a measure of the number of argument values, where an argument value of **long** or **double** type 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 selected method's descriptor. The redundancy is historical.

The zero byte (following ***count***) exists to reserve space for an additional operand used in certain Sun implementations, which replace **invokeinterface** by a specialized pseudo-instruction at runtime. 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 **long** and **double** types must store in two consecutive local variables, thus more than ***nargs*** local variables may be required to pass ***nargs*** argument values to the invoked method.

**Mnemonic:** **invokespecial**

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

**Opcode:** 183 (0xB7)

**Operands:** ***indexbyte1***, ***indexbyte2***

**Operand Stack:** Before: …, ***objectref***, [***arg1***, [***arg2*** …]] >>>
After: …

**Description:** The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index's value is (***indexbyte1***<<8)| ***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to a method, which gives the name and descriptor of the method as well as a symbolic reference to the class in which the method is to be found. The named method resolves. Finally, if the resolved method is **protected**, and it is either a member of the current class or a member of a superclass of the current class, ***objectref***'s class must be either the current class or a subclass of the current class.

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

1) The **ACC_SUPER** flag is set for the current class.

2) The resolved method's class is a superclass of the current class.

3) The resolved method is not an instance initialization method.

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:

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

2) 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.

3) Otherwise, an **AbstractMethodError** is raised.

***objectref*** must be of **reference** type 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 selected instance method's descriptor.

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

If the method is not **native**, the ***nargs*** argument values and ***objectref*** pop from the operand stack. A new stack frame creates on the JVM stack for the method being invoked. The ***objectref*** and argument values are consecutively made the values of local variables of the new stack frame, with ***objectref*** in local variable 0, ***arg1*** in local variable 1 (or, if ***arg1*** is of **long** or **double** type, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion prior to being stored in a local variable. The new stack frame is then made current, and the JVM **pc** register 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 into the JVM, that bounding is done. The ***nargs*** argument values and ***objectref*** pop from the operand stack and pass as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion prior to being passed as a parameter. The parameters pass and the code invokes in an implementation-dependent manner. When the platform-dependent code returns:

1) If the **native** method is synchronized, the monitor associated with ***objectref*** releases or exits as if by execution of a **monitorexit** instruction.

2) If the **native** method returns a value, the return value of the platform-dependent code converts in an implementation-dependent way to the return type of the **native** method and pushes to 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 of the JVM specification can be thrown.

This instruction throws a **NoSuchMethodError** if the resolved method is an instance initialization method, and the class in which it declares is not the class symbolically referenced by the instruction.

This instruction throws an **IncompatibleClassChangeError** if the resolved method is a class (**static**) method.

This instruction throws an **AbstractMethodError** if no method matching the resolved name and descriptor selects, or if the selected method is **abstract**.

**Runtime Exceptions:**

This instruction throws a **NullPointerException** if ***objectref*** is **null**.

This instruction throws an **UnsatisfiedLinkError** if the selected method is **native** and the code that implements the method cannot be bound.

**Notes:**

The difference between **invokespecial** and **invokevirtual** is that **invokevirtual** invokes a method based on the object's class. The **invokespecial** instruction invokes instance initialization methods as well as **private** methods and methods of the current class's superclass.

**invokespecial** 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 **long** and **double** types must store in two consecutive local variables, thus more than ***nargs*** local variables may be required to pass ***nargs*** argument values to the invoked method.

**Mnemonic:** **invokestatic**

**Operation:** Invoke class (**static**) method.

**Opcode:** 184 (0xB8)

**Operands:** ***indexbyte1***, ***indexbyte2***

**Operand Stack:** Before: …, [***arg1***, [***arg2*** …]] >>>
After: …

**Description:** The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index's value is (***indexbyte1***<<8)| ***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to a method, which gives the name and descriptor of the method as well as a symbolic reference to the class in which the method is to be found. The named method resolves. The method must not be the class or interface initialization method. It must be **static**, and therefore cannot be **abstract**.

On successful method resolution, the class declaring the resolved method initializes (unless already initialized).

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

If the method is **synchronized**, the monitor associated with the resolved class is acquired or rentered.

If the method is not **native**, the ***nargs*** argument values pop from the operand stack. A new stack frame creates on the JVM stack for the method being invoked. The ***nargs*** argument values are consecutively made the values of local variables of the new stack frame, with ***arg1*** in local variable 0 (or, if ***arg1*** is of **long** or **double** type, in local variables 0 and 1), and so on. Any argument value that is of a floating-point type undergoes value set conversion prior to being stored in a local variable. The new stack frame is then made current, and the JVM **pc** register 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 into the JVM, that bounding is done. The ***nargs*** argument values pop from the operand stack and pass as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion prior to being passed as a parameter. The parameters pass and the code invokes in an implementation-dependent manner. When the platform-dependent code returns:

1) If the **native** method is synchronized, the monitor associated with the resolved class releases or exits as if by execution of a **monitorexit** instruction.

2) If the **native** method returns a value, the return value of the platform-dependent code converts in an implementation-dependent way to the return type of the **native** method and pushes to 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 of the JVM specification can be thrown.<br><br>This instruction throws an **IncompatibleClassChangeError** if the resolved method is an instance method. |
| **Runtime Exceptions:** | If execution of this instruction causes initialization of the referenced class, the instruction may throw an **Error** as detailed in section 2.17.5 of the JVM specification.<br><br>This instruction throws an **UnsatisfiedLinkError** if the resolved method is **native** and the code that implements the method cannot be bound. |
| **Notes:** | The ***nargs*** argument values are not one-to-one with the first ***nargs*** local variables. Argument values of **long** and **double** types must store in two consecutive local variables, thus more than ***nargs*** local variables may be required to pass ***nargs*** argument values to the invoked method. |

**Mnemonic:** **invokevirtual**

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

**Opcode:** 182 (0xB6)

**Operands:** ***indexbyte1***, ***indexbyte2***

**Operand Stack:** Before: …, ***objectref***, [***arg1***, [***arg2*** …]] >>>
After: …

**Description:** The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index's value is (***indexbyte1***<<8)| ***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to a method, which gives the name and descriptor of the method as well as a symbolic reference to the class in which the method is to be found. The named method resolves. The method must not be an instance initialization method or the class or interface initialization method. Finally, if the resolved method is **protected**, and it is either a member of the current class or a member of a superclass of the current class, ***objectref***'s class must be either the current class or a subclass of the current class.

Let *C* be ***objectref***'s class. The actual method to be invoked is selected by the following procedure:

1) If *C* contains a declaration for an instance method with the same name and descriptor as the resolved method, and the resolved method is accessible from *C*, this is the method to be invoked, and the lookup procedure terminates.

2) 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.

3) Otherwise, an **AbstractMethodError** is raised.

***objectref*** must be of **reference** type 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 selected instance method's descriptor.

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

If the method is not **native**, the ***nargs*** argument values and ***objectref*** pop from the operand stack. A new stack frame creates on the JVM stack for the method being invoked. The ***objectref*** and argument values are consecutively made the values of local variables of the new stack frame, with ***objectref*** in local variable 0, ***arg1*** in local variable 1 (or, if ***arg1*** is of **long** or **double** type, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion prior to being stored in a local variable. The new stack frame is then made current, and the JVM **pc** register 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 into the JVM, that bounding is done. The ***nargs*** argument values and ***objectref*** pop from the operand stack and pass as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion prior to being passed as a parameter. The parameters pass and the code invokes in an implementation-dependent manner. When the platform-dependent code returns:

1) If the **native** method is synchronized, the monitor associated with ***objectref*** releases or exits as if by execution of a **monitorexit** instruction.

2) If the **native** method returns a value, the return value of the platform-dependent code converts in an implementation-dependent way to the return type of the **native** method and pushes to 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 of the JVM specification can be thrown.

This instruction throws an **IncompatibleClassChangeError** if the resolved method is a class (**static**) method.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***objectref*** is **null**.

This instruction throws an **AbstractMethodError** if no method matching the resolved name and descriptor selects, or if the selected method is **abstract**.

This instruction throws an **UnsatisfiedLinkError** if the selected method is **native** and the code that implements the method cannot be bound.

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

**Mnemonic:** **ior**

**Operation:** Boolean OR **int**.

**Opcode:** 128 (0x80)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. An **int** ***result*** calculates by taking the bitwise inclusive OR of ***value1*** and ***value2***. The result pushes to the operand stack.

**Mnemonic:** **irem**

**Operation:** Remainder **int**.

**Opcode:** 112 (0x70)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. The **int** ***result*** from ***value1***-(***value1***/***value2***)*****value2*** pushes to the operand stack.

The result of this 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 result's magnitude is always less than the divisor's magnitude.

**Runtime Exceptions:** This instruction throws an **ArithmeticException** if the divisor's value is 0.

**Mnemonic:** **ireturn**

**Operation:** Return **int** from method.

**Opcode:** 172 (0xAC)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: [empty]

**Description:** The current method's return type must be one of **boolean**, **byte**, **char**, **int**, or **short**. ***value***'s type must be **int**. If the current method is **synchronized**, the monitor acquired or reentered on that method's invocation is released or exited (respectively) as if by executing the **monitorexit** instruction. If no exception is thrown, ***value*** pops from the current stack frame's operand stack and pushes to the operand stack of the invoker's stack frame. Any other values on the current method's operand stack discard.

The interpreter returns control to the current method's invoker, reinstating the invoker's stack frame as the current stack frame.

**Runtime Exceptions:** This instruction throws an **IllegalMonitorStateException** 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.

This instruction also throws an **IllegalMonitorStateException** if the JVM implementation enforces the rules on structured lock usage, and the number of lock operations performed by a thread on a lock does not match the number of unlock operations by that thread on that lock (whether the method invocation completes normally or abruptly).

**Mnemonic:** **ishl**

**Operation:** Shift left **int**.

**Opcode:** 120 (0x78)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. An **int** ***result*** calculates by shifting ***value1*** left by ***s*** bit positions, where ***s*** is the value of the low 5 bits of ***value2***. The ***result*** pushes to 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.

**Mnemonic:** **ishr**

**Operation:** Arithmetic shift right **int**.

**Opcode:** 122 (0x7A)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. An **int** ***result*** calculates by shifting ***value1*** right by ***s*** bit positions, with sign extension, where ***s*** is the value of the low 5 bits of ***value2***. The ***result*** pushes to the operand stack.

**Notes:** The resulting value is floor ***value1***/**2**^***s***, where ***s*** is ***value2***&0x1F. For nonnegative ***value1***, this is equivalent to truncating **int** division 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.

**Mnemonic:** **istore**

**Operation:** Store **int** into local variable.

**Opcode:** 54 (0x36)

**Operands:** ***index***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***index*** is an unsigned byte that must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **int** type. It pops from the operand stack, and the value of the local variable at ***index*** sets to ***value***.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **istore_0**

**Operation:** Store **int** into local variable 0.

**Opcode:** 59 (0x3B)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 0 must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **int** type. It pops from the operand stack, and the value of the local variable at 0 sets to ***value***.

**Notes:** This instruction is the same as **istore** (with a 0 index operand), except that the 0 in **istore_0** is implicit.

**Mnemonic:** **istore_1**

**Operation:** Store **int** into local variable 1.

**Opcode:** 60 (0x3C)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 1 must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **int** type. It pops from the operand stack, and the value of the local variable at 1 sets to ***value***.

**Notes:** This instruction is the same as **istore** (with a 1 index operand), except that the 1 in **istore_1** is implicit.

**Mnemonic:** **istore_2**

**Operation:** Store **int** into local variable 2.

**Opcode:** 61 (0x3D)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 2 must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **int** type. It pops from the operand stack, and the value of the local variable at 2 sets to ***value***.

**Notes:** This instruction is the same as **istore** (with a 2 index operand), except that the 2 in **istore_2** is implicit.

**Mnemonic:** **istore_3**

**Operation:** Store **int** into local variable 3.

**Opcode:** 62 (0x3E)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 3 must be an index into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **int** type. It pops from the operand stack, and the value of the local variable at 3 sets to ***value***.

**Notes:** This instruction is the same as **istore** (with a 3 index operand), except that the 3 in **istore_3** is implicit.

**Mnemonic:** **isub**

**Operation:** Subtract **int**.

**Opcode:** 100 (0x64)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. The **int** ***result*** is ***value1***-***value2***, which pushes to the operand stack.

For **int** subtraction, it is always the case that ***a***-***b*** results in the same value 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 **int** type. If overflow occurs, the result's sign may not be the same as the sign of the mathematical sum of both values.

Despite the fact that overflow may occur, execution of this instruction never throws a runtime exception.

**Mnemonic:** **iushr**

**Operation:** Logical shift right **int**.

**Opcode:** 124 (0x7C)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. An **int** ***result*** calculates by shifting ***value1*** right by ***s*** bit positions, with zero extension, where ***s*** is the value of ***value2***'s low 5 bits. The result pushes to the operand stack.

**Notes:** If ***value1*** is positive and ***s*** is ***value2***&0x1F, 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***)+(2<<~***s***). The addition of the (2<<~***s***) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive.

**Mnemonic:** **ixor**

**Operation:** Boolean XOR **int**.

**Opcode:** 130 (0x82)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **int** type. Those values pop from the operand stack. An **int** ***result*** calculates by taking the bitwise exclusive OR of ***value1*** and ***value2***. The result pushes to the operand stack.

**Mnemonic:** **jsr**

**Operation:** Jump subroutine.

**Opcode:** 168 (0xA8)

**Operands:** ***branchbyte1***, ***branchbyte2***

**Operand Stack:** Before: … >>>
After: …, ***address***

**Description:** The ***address*** of the opcode of the instruction immediately following this instruction pushes to the operand stack as a value of **returnAddress** type. Unsigned bytes ***branchbyte1*** and ***branchbyte2*** construct into a 16-bit ***branchoffset***, via (***branchbyte1***<<8)|***branchbyte2***. Execution proceeds at ***branchoffset*** from the address of this instruction. The target address must be that of an instruction opcode within the method that contains the **jsr** instruction.

**Notes:** This instruction is used with the **ret** instruction to implement Java language **finally** clauses. This instruction pushes the address to the operand stack and **ret** retrieves that address from a local variable. That asymmetry is intentional.

**Mnemonic:** **jsr_w**

**Operation:** Jump subroutine (wide index).

**Opcode:** 201 (0xC9)

**Operands:** ***branchbyte1***, ***branchbyte2***, ***branchbyte3***, ***branchbyte4***

**Operand Stack:** Before: … >>>
After: …, ***address***

**Description:** The ***address*** of the opcode of the instruction immediately following this instruction pushes to the operand stack as a value of **returnAddress** type. Unsigned bytes ***branchbyte1***, ***branchbyte2***, ***branchbyte3***, and ***branchbyte4*** construct into a 32-bit ***branchoffset***, via (***branchbyte1***<<24)|(***branchbyte2***<<16)|(***branchbyte3***<<8)|***branchbyte4***. Execution proceeds at ***branchoffset*** from the address of this instruction. The target address must be that of an instruction opcode within the method that contains the **jsr_w** instruction.

**Notes:** This instruction is used with the **ret** instruction to implement Java language **finally** clauses. This instruction pushes the address to the operand stack and **ret** retrieves that address from a local variable. That asymmetry is intentional.

Despite the 32-bit branch address, other factors may limit the size of a method to 65535 bytes. Future releases of the JVM may raise this limit.

**Mnemonic:** **l2d**

**Operation:** Convert **long** to **double**.

**Opcode:** 138 (0x8A)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **long** type. It pops from the operand stack and converts to a **double** ***result*** (using IEEE 754 round to nearest mode), which pushes to the operand stack.

**Notes:** This instruction performs a widening primitive conversion that may lose precision because **double** values have only 53 significand bits.

**Mnemonic:** **l2f**

**Operation:** Convert **long** to **float**.

**Opcode:** 137 (0x89)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **long** type. It pops from the operand stack and converts to a **float** ***result*** (using IEEE 754 round to nearest mode), which pushes to the operand stack.

**Notes:** This instruction performs a widening primitive conversion that may lose precision because **float** values have only 24 significand bits.

**Mnemonic:** **l2i**

**Operation:** Convert **long** to **int**.

**Opcode:** 136 (0x88)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value***, which is on top of the operand stack, must be of **long** type. It pops from the operand stack and converts to an **int** ***result*** (by discarding the upper 32 bits), which pushes to the operand stack.

**Notes:** This instruction performs a narrowing primitive conversion. As a result, information about ***value***'s overall magnitude may be lost. Furthermore, ***result*** and ***value*** may have different signs.

**Mnemonic:** **ladd**

**Operation:** Add **long**.

**Opcode:** 97 (0x61)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack. The **long** ***result*** from ***value1***+***value2*** pushes to 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 **long** type. If overflow occurs, the result's sign may not be the same as the sign of the mathematical sum of both values.

This instruction never throws an exception, even though overflow may occur.

**Mnemonic:** **laload**

**Operation:** Load **long** from array.

**Opcode:** 47 (0x2F)

**Operands:** None

**Operand Stack:** Before: …, ***arrayref***, ***index*** >>>
After: …, ***value***

**Description:** ***arrayref*** must be of **reference** type and must refer to an array whose components are of **long** type. Furthermore, ***index*** must be of **int** type. Both ***arrayref*** and ***index*** pop from the operand stack. The **long** ***value*** in the array component at ***index*** retrieves and pushes to the operand stack.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***arrayref*** is **null**.

This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array.

**Mnemonic:** **land**

**Operation:** Boolean AND **long**.

**Opcode:** 127 (0x7F)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack. The **long** ***result*** from a bitwise AND (conjunction) of both values pushes to the operand stack.

| | |
|---|---|
| **Mnemonic:** | **lastore** |
| **Operation:** | Store into **long** array. |
| **Opcode:** | 80 (0x50) |
| **Operands:** | None |
| **Operand Stack:** | Before: …, ***arrayref***, ***index***, ***value*** >>><br>After: … |
| **Description:** | ***arrayref*** must be of **reference** type and must refer to an array whose components are of **long** type. Furthermore, ***index*** must be of **int** type and ***value*** must both be of **long** type. ***arrayref***, ***index***, and ***value*** pop from the operand stack. The **long** ***value*** stores as the array component at ***index***. |
| **Runtime Exceptions:** | This instruction throws a **NullPointerException** if ***arrayref*** is **null**.<br><br>This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array. |

**Mnemonic:** **lcmp**

**Operation:** Compare **long**.

**Opcode:** 148 (0x94)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack and a signed integer comparison occurs. If ***value1*** is greater than ***value2***, the **int** value 1 pushes to the operand stack. If ***value1*** equals ***value2***, the **int** value 0 pushes to the operand stack. If ***value1*** is less than ***value2***, the **int** value -1 pushes to the operand stack.

**Mnemonic:** **lconst_0**

**Operation:** Push **long** constant **0L**.

**Opcode:** 9 (0x09)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **0**

**Description:** Push the **long** constant 0 to the operand stack.

**Mnemonic:** **lconst_1**

**Operation:** Push **long** constant **1L**.

**Opcode:** 10 (0x0A)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, **1**

**Description:** Push the **long** constant 1 to the operand stack.

**Mnemonic:** **ldc**

**Operation:** Push item from runtime constant pool.

**Opcode:** 18 (0x12)

**Operands:** ***index***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** ***index*** is an unsigned byte that must be a valid index into the current class's runtime constant pool. The pool's entry at ***index*** must be a symbolic reference to a class, a string literal, or a runtime constant of type **int** or **float**.

If the runtime constant pool entry is a symbolic reference to a class, the named class is resolved and a reference to the **Class** object representing that class, ***value***, pushes to the operand stack.

If the runtime constant pool entry is a reference to an instance of class **String** representing a string literal, a **reference** to that instance, ***value***, pushes to the operand stack.

If the runtime constant pool entry is a runtime constant of type **int** or **float**, that constant's **int** or **float** ***value*** pushes to 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 of the JVM specification can be thrown.

**Notes:** This instruction can only push a **float** value from the float value set because a **float** constant in the constant pool must be taken from the float value set.

This instruction's description has been updated to reflect Java 5 changes.

**Mnemonic:** **ldc_w**

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

**Opcode:** 19 (0x13)

**Operands:** ***indexbyte1***, ***indexbyte1***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** Assemble unsigned ***indexbyte1*** and ***indexbyte2*** into an unsigned 16-bit index into the current class's runtime constant pool, via (***indexbyte1***<<8)|***indexbyte2***. The resulting index must be a valid index into the current class's runtime constant pool. The pool's entry at the index must be a symbolic reference to a class, a string literal, or a runtime constant of type **int** or **float**.

If the runtime constant pool entry is a symbolic reference to a class, the named class is resolved and a reference to the **Class** object representing that class, ***value***, pushes to the operand stack.

If the runtime constant pool entry is a reference to an instance of class **String** representing a string literal, a **reference** to that instance, ***value***, pushes to the operand stack.

If the runtime constant pool entry is a runtime constant of type **int** or **float**, that constant's **int** or **float** ***value*** pushes to 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 of the JVM specification can be thrown.

**Notes:** This instruction is identical to **ldc**, except for **ldc_w**'s wider runtime constant pool index.

This instruction can only push a **float** value from the float value set because a **float** constant in the constant pool must be taken from the float value set.

This instruction's description has been updated to reflect Java 5 changes.

**Mnemonic:** **ldc2_w**

**Operation:** Push **long** or **double** from runtime constant pool (wide index).

**Opcode:** 20 (0x14)

**Operands:** ***indexbyte1***, ***indexbyte1***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** Assemble unsigned ***indexbyte1*** and ***indexbyte2*** into an unsigned 16-bit index into the current class's runtime constant pool, via (***indexbyte1***<<8)|***indexbyte2***. The resulting index must be a valid index into the current class's runtime constant pool. The pool's entry at ***index*** must be a runtime constant of type **long** or **double**. That constant's **long** or **double** ***value*** pushes to the operand stack.

**Notes:** This instruction does not have a matching **ldc2** instruction (which would use a narrower single-byte index).

This instruction can only push a **double** value from the double value set because a **double** constant in the constant pool must be taken from the double value set.

**Mnemonic:** **ldiv**

**Operation:** Divide **long**.

**Opcode:** 109 (0x6D)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack. The **long** ***result*** from ***value1***/***value2*** pushes to the operand stack.

This division rounds towards zero: the quotient produced for **long** values in ***n***/***d*** is a **long** value ***q*** whose magnitude is as large as possible while satisfying |***d***.***q***| <= |***n***|. ***q*** is positive when |***n***|>=|***d***| and ***n*** and ***d*** have the same sign. ***q*** is negative when |***n***|>=|***d***| and ***n*** and ***d*** have opposite signs.

One special case 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, overflow occurs and the result equals the dividend. No exception is thrown in this case.

**Runtime Exceptions:** This instruction throws an **ArithmeticException** if the divisor's value is 0.

**Mnemonic:** **lload**

**Operation:** Load **long** from local variable.

**Opcode:** 22 (0x16)

**Operands:** ***index***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** ***index*** is an unsigned byte. Both ***index*** and ***index***+1 must be indices into the current stack frame's local variable array. Furthermore, the local variable at ***index*** must contain a **long**. The ***value*** of the local variable at ***index*** pushes to the operand stack.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **lload_0**

**Operation:** Load **long** from local variable 0.

**Opcode:** 30 (0x1E)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 0 and 1 must be indices into the current stack frame's local variable array. Furthermore, the local variable at 0 must contain a **long**. The ***value*** of the local variable at 0 pushes to the operand stack.

**Notes:** This instruction is the same as **lload** (with a 0 index operand), except that the 0 in **lload_0** is implicit.

**Mnemonic:** **lload_1**

**Operation:** Load **long** from local variable 1.

**Opcode:** 31 (0x1F)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 1 and 2 must be indices into the current stack frame's local variable array. Furthermore, the local variable at 1 must contain a **long**. The ***value*** of the local variable at 1 pushes to the operand stack.

**Notes:** This instruction is the same as **lload** (with a 1 index operand), except that the 1 in **lload_1** is implicit.

**Mnemonic:** **lload_2**

**Operation:** Load **long** from local variable 2.

**Opcode:** 32 (0x20)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 2 and 3 must be indices into the current stack frame's local variable array. Furthermore, the local variable at 2 must contain a **long**. The ***value*** of the local variable at 2 pushes to the operand stack.

**Notes:** This instruction is the same as **lload** (with a 2 index operand), except that the 2 in **lload_2** is implicit.

**Mnemonic:** **lload_3**

**Operation:** Load **long** from local variable 3.

**Opcode:** 33 (0x21)

**Operands:** None

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** 3 and 4 must be indices into the current stack frame's local variable array. Furthermore, the local variable at 3 must contain a **long**. The ***value*** of the local variable at 3 pushes to the operand stack.

**Notes:** This instruction is the same as **lload** (with a 3 index operand), except that the 3 in **lload_3** is implicit.

**Mnemonic:** **lmul**

**Operation:** Multiply **long**.

**Opcode:** 105 (0x69)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack. The **long** ***result*** from ***value1*******value2*** pushes to the operand stack.

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

This instruction never throws an exception, even though overflow may occur.

**Mnemonic:** **lneg**

**Operation:** Negate **long**.

**Opcode:** 117 (0x75)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …, ***result***

**Description:** ***value*** must be of **long** type. It pops from the operand stack. The **long** ***result*** is the arithmetic negation of ***value***, -***value***, which pushes to the operand stack.

Negation is the same as subtraction from zero, for **long** values. Because the JVM uses two's-complement representation for integers and the range of two's-complement values is asymmetric, the negation of the maximum negative long integer results in the same maximum negative long integer. Despite this overflow, no exception is thrown.

For all **long** values ***x***, -***x*** equals (~***x***)+1.

**Mnemonic:** **lookupswitch**

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

**Opcode:** 171 (0xAB)

**Operands:** ***<0-3 byte pad>***, ***defaultbyte1***, ***defaultbyte2***, ***defaultbyte3***, ***defaultbyte4***, ***npairs1***, ***npairs2***, ***npairs3***, ***npairs4***, ***match-offset pairs…***

**Operand Stack:** Before: …, ***key*** >>>
After: …

**Description:** This instruction is variable-length. Immediately after its opcode, between 0 and 3 null bytes (zero bytes, not the null object) insert as padding. The number of null bytes is chosen so that ***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 follows a series of signed 32-bit values: ***default***, ***npairs***, and then ***npairs*** pairs of signed 32-bit values. ***npairs*** must be greater than or equal to 0. Each of the ***npairs*** pairs consists of an **int** ***match*** and a signed 32-bit ***offset***. Each of these signed 32-bit values constructs as (***byte1***<<24)|(***byte2***<<16)|(***byte3***<<8)|***byte4***.

The ***match-offset*** pairs table must be sorted in increasing numerical order of ***match***.

***key*** must be of **int** type and pops from the operand stack. ***key*** compares against ***match*** values. If it equals one of them, a target address calculates by adding the corresponding ***offset*** to the address of this instruction's opcode. If ***key*** doesn't match any ***match*** value, the target address calculates by adding ***default*** to the address of this instruction's opcode. Execution 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 the **lookupswitch** instruction.

**Notes:** The alignment required of the 4-byte operands of this instruction guarantees 4-byte alignment of those operands if and only if the method containing this instruction starts on a 4-byte boundary.

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

**Mnemonic:** **lor**

**Operation:** Boolean OR **long**.

**Opcode:** 129 (0x81)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack. A **long** ***result*** calculates by taking the bitwise inclusive OR of ***value1*** and ***value2***. The result pushes to the operand stack.

**Mnemonic:** **lrem**

**Operation:** Remainder **long**.

**Opcode:** 113 (0x71)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack. The **long** ***result*** from ***value1***-(***value1***/***value2***)*****value2*** pushes to the operand stack.

The result of this 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 result's magnitude is always less than the divisor's magnitude.

**Runtime Exceptions:** This instruction throws an **ArithmeticException** if the divisor's value is 0.

**Mnemonic:** **lreturn**

**Operation:** Return **long** from method.

**Opcode:** 173 (0xAD)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: [empty]

**Description:** Both the current method's return type and ***value***'s type must be **long**. If the current method is **synchronized**, the monitor acquired or reentered on that method's invocation is released or exited (respectively) as if by executing the **monitorexit** instruction. If no exception is thrown, ***value*** pops from the current stack frame's operand stack and pushes to the operand stack of the invoker's stack frame. Any other values on the current method's operand stack discard.

The interpreter returns control to the current method's invoker, reinstating the invoker's stack frame as the current stack frame.

**Runtime Exceptions:** This instruction throws an **IllegalMonitorStateException** 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.

This instruction also throws an **IllegalMonitorStateException** if the JVM implementation enforces the rules on structured lock usage, and the number of lock operations performed by a thread on a lock does not match the number of unlock operations by that thread on that lock (whether the method invocation completes normally or abruptly).

**Mnemonic:** **lshl**

**Operation:** Shift left **long**.

**Opcode:** 121 (0x79)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** must be of **long** type and ***value2*** must be of **int** type. Those values pop from the operand stack. A **long** ***result*** calculates by shifting ***value1*** left by ***s*** bit positions, where ***s*** is the value of the low 6 bits of ***value2***. The ***result*** pushes to 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 63, inclusive, as if ***value2*** were subjected to a bitwise logical AND with the mask value 0x3F.

**Mnemonic:** **lshr**

**Operation:** Arithmetic shift right **long**.

**Opcode:** 123 (0x7B)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** must be of **long** type and ***value2*** must be of **int** type. Those values pop from the operand stack. A **long** ***result*** calculates 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*** pushes to the operand stack.

**Notes:** The resulting value is floor ***value1***/2^***s***, 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 always in the range 0 to 63, inclusive, as if ***value2*** were subjected to a bitwise logical AND with the mask value 0x3F.

**Mnemonic:** **lstore**

**Operation:** Store **long** into local variable.

**Opcode:** 55 (0x37)

**Operands:** ***index***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** ***index*** is an unsigned byte. Both ***index*** and ***index***+1 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **long** type. It pops from the operand stack, and the value of the local variables at ***index*** and ***index***+1 set to ***value***.

**Notes:** This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **lstore_0**

**Operation:** Store **long** into local variable 0.

**Opcode:** 63 (0x3F)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 0 and 1 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **long** type. It pops from the operand stack and stores in the local variables at 0 and 1.

**Notes:** This instruction is the same as **lstore** (with a 0 index operand), except that the 0 in **lstore_0** is implicit.

**Mnemonic:** **lstore_1**

**Operation:** Store **long** into local variable 1.

**Opcode:** 64 (0x40)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 1 and 2 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **long** type. It pops from the operand stack and stores in the local variables at 1 and 2.

**Notes:** This instruction is the same as **lstore** (with a 1 index operand), except that the 1 in **lstore_1** is implicit.

**Mnemonic:** **lstore_2**

**Operation:** Store **long** into local variable 2.

**Opcode:** 65 (0x41)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 2 and 3 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **long** type. It pops from the operand stack and stores in the local variables at 2 and 3.

**Notes:** This instruction is the same as **lstore** (with a 2 index operand), except that the 2 in **lstore_2** is implicit.

**Mnemonic:** **lstore_3**

**Operation:** Store **long** into local variable 3.

**Opcode:** 66 (0x42)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** 3 and 4 must be indices into the current stack frame's local variable array. Furthermore, the ***value*** on top of the operand stack must be of **long** type. It pops from the operand stack and stores in the local variables at 3 and 4.

**Notes:** This instruction is the same as **lstore** (with a 3 index operand), except that the 3 in **lstore_3** is implicit.

**Mnemonic:** **lsub**

**Operation:** Subtract **long**.

**Opcode:** 101 (0x65)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** Both ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack. The **long** ***result*** is ***value1*****-*****value2***, which pushes to the operand stack.

For **long** subtraction, it is always the case that ***a*****-*****b*** results in the same value as ***a*** +(-***b***). For **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 **long** type. If overflow occurs, the result's sign may not be the same as the sign of the mathematical sum of both values.

Despite the fact that overflow may occur, execution of this instruction never throws a runtime exception.

**Mnemonic:** **lushr**

**Operation:** Logical shift right **long**.

**Opcode:** 125 (0x7D)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** must be of **long** type and ***value2*** must be of **int** type. Those values pop from the operand stack. A **long** ***result*** calculates by shifting ***value1*** right by ***s*** bit positions, with zero extension, where ***s*** is the value of ***value2***'s low 6 bits. The result pushes to 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.

**Mnemonic:** **lxor**

**Operation:** Boolean XOR **long**.

**Opcode:** 131 (0x83)

**Operands:** None

**Operand Stack:** Before: …, ***value1***, ***value2*** >>>
After: …, ***result***

**Description:** ***value1*** and ***value2*** must be of **long** type. Those values pop from the operand stack. A **long** ***result*** calculates by taking the bitwise exclusive OR of ***value1*** and ***value2***. The result pushes to the operand stack.

**Mnemonic:** **monitorenter**

**Operation:** Enter monitor for object.

**Opcode:** 194 (0xC2)

**Operands:** None

**Operand Stack:** Before: …, ***objectref*** >>>
After: …

**Description:** ***objectref*** must be of **reference** type.

Each object has an associated monitor. The thread that executes this instruction gains ownership of the monitor that associates with ***objectref***. If another thread already owns the monitor associated with ***objectref***, the current thread waits until the object unlocks, and then tries again to gain ownership. If the current thread already owns the monitor that associates 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 monitor's owner and sets the monitor's entry count to 1.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***objectref*** is **null**.

**Mnemonic:** **monitorexit**

**Operation:** Exit monitor for object.

**Opcode:** 195 (0xC3)

**Operands:** None

**Operand Stack:** Before: …, ***objectref*** >>>
After: …

**Description:** ***objectref*** must be of **reference** type.

The current thread should be the owner of the monitor associated with the ***objectref*** -referenced instance. 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 waiting to acquire that monitor are allowed to attempt to do so.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***objectref*** is **null**.

This instruction throws an **IllegalMonitorStateException** if the current thread is not the owner of the monitor.

This instruction also throws an **IllegalMonitorStateException** if the JVM implementation enforces the rules on structured lock usage, and the number of unlock operations performed by a thread on a lock exceeds the number of lock operations performed by the thread on that lock since the method invocation.

**Mnemonic:** **multianewarray**

**Operation:** Create new multidimensional array.

**Opcode:** 197 (0xC5)

**Operands:** ***indexbyte1***, ***indexbyte2***, ***dimensions***

**Operand Stack:** Before: …, ***count1***, [***count2***, …] >>>
After: …, ***arrayref***

**Description:** ***dimensions*** 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 **int** type, and must be nonnegative. ***count1*** is the desired length in the first dimension, ***count2*** is the desired length in the second dimension, and so on.

All ***count*** values pop from the operand stack. The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index value is (***indexbyte1***<<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 resolves. The resulting entry must be an array class type of dimensionality greater than or equal to ***dimensions***.

A new multidimensional array of the array type allocates from the garbage-collected heap. If any ***count*** value is zero, no subsequent dimensions allocate. The components of the array in the first dimension initialize to subarrays of the second dimension's type, and so on. The components of the last allocated array dimension initialize to the default initial value for the components type. A **reference** ***arrayref*** to the new array pushes to 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 of the JVM specification can be thrown.

This instruction throws an **IllegalAccessError** if the current class does not have permission to access the element type of the resolved array class.

**Runtime Exceptions:** This instruction throws a **NegativeArraySizeException** if any of the ***dimensions*** values are less than zero.

**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*** of the **multianewarray** instruction. In that case, only the first ***dimensions*** of the array create.

**Mnemonic:** **new**

**Operation:** Create new object.

**Opcode:** 187 (0xBB)

**Operands:** ***indexbyte1***, ***indexbyte2***

**Operand Stack:** Before: … >>>
After: …, ***objectref***

**Description:** The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index value is (***indexbyte1***<<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 resolves and should result in a class type (it should not result in an array or interface type). Memory for a new instance of that class allocates from the garbage-collected heap, and the instance variables of the new object initialize to their default initial values. The ***objectref***, a **reference** to the instance, pushes to the operand stack.

On successful resolution of the class, it initializes if not already 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 of the JVM specification can be thrown.

This instruction throws an **InstantiationError** if the symbolic reference to the class, array, or interface type resolves to an interface or is an **abstract** class.

**Runtime Exceptions:** If execution of this instruction causes initialization of the referenced class or interface, the instruction may throw an **Error** as detailed in section 2.17.5 of the JVM specification.

**Notes:** This 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.

**Mnemonic:** **newarray**

**Operation:** Create new array.

**Opcode:** 188 (0xBC)

**Operands:** ***atype***

**Operand Stack:** Before: …, ***count*** >>>
After: …, ***arrayref***

**Description:** ***count***, which must be of **int** type, pops from the operand stack and represents the number of array components to be created.

***atype*** is a code that indicates the type of array to create: 4 (**T_BOOLEAN**), 5 (**T_CHAR**), 6 (**T_FLOAT**), 7 (**T_DOUBLE**), 8 (**T_BYTE**), 9 (**T_SHORT**), 10 (**T_INT**), 11 (**T_LONG**).

A new array of ***count*** length and whose components are of ***atype*** type allocates from the garbage-collected heap. A **reference** ***arrayref*** to this new array object pushes to the operand stack. Each of the new array's components initializes to the default initial value of the array's type.

**Runtime Exceptions:** This instruction throws a **NegativeArraySizeException** if ***count*** is less than zero.

**Notes:** Sun's JVM implementation implements **boolean** arrays as arrays of 8-bit values.

**Mnemonic:** **nop**

**Operation:** No operation.

**Opcode:** 0 (0x00)

**Operands:** None

**Operand Stack:** Unchanged

**Description:** Don’t do anything.

**Mnemonic:** **pop**

**Operation:** Pop the top operand stack value.

**Opcode:** 87 (0x57)

**Operands:** None

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** Pop the top value from the operand stack.

This instruction must not be used if ***value*** is of **long** or **double** type.

**Mnemonic:** **pop2**

**Operation:** Pop the top one or two operand stack values.

**Opcode:** 88 (0x58)

**Operands:** None

**Operand Stack:** Before: …, ***value2***, ***value1*** >>>
After: …

***value1*** and ***value2*** must not be of type **double** or **long**.

Before: …, ***value*** >>>
After: …

***value*** must be of type **double** or **long**.

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

**Mnemonic:** **putfield**

**Operation:** Set field in object.

**Opcode:** 181 (0xB5)

**Operands:** ***indexbyte1***, ***indexbyte2***

**Operand Stack:** Before: …, ***objectref***, ***value*** >>>
After: …

**Description:** The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index value is (***indexbyte1***<<8)|***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to a field, 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. ***objectref***'s class must not be an array. If the field is **protected** and a member of a superclass of the current class, and the field is not declared in the same run-time package as the current class, ***objectref***'s class must be either the current class or a subclass of the current class.

The referenced field resolves. ***value***'s type must be compatible with the referenced field's descriptor. If the field descriptor's type is **boolean**, **byte**, **char**, **int**, or **short**, ***value*** must be an **int**. If the field descriptor's type is **double**, **float**, or **long**, ***value*** must be a **double**, **float**, or **long**, respectively. If the field descriptor's type is a reference type, ***value*** must be of a type that is assignment compatible with the field descriptor type. If the field is **final**, it should be declared in the **<init>** method of the current class and ***objectref*** must be the **this** argument of the **<init>** method. Otherwise, an **IllegalAccessError** throws.

***value*** and ***objectref*** pop from the operand stack. ***objectref*** must be of **reference** type. ***value*** undergoes value set conversion, resulting in ***value'***, and the referenced field in ***objectref*** sets 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 of the JVM specification can be thrown.

This instruction throws an **IncompatibleClassChangeError** if the resolved field is a **static** field.

This instruction throws an **IllegalAccessError** if the field is **final** but not declared in the **<init>** method of the current class, or ***objectref*** is not the **this** argument of the **<init>** method.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***objectref*** is **null**.

**Notes:** This instruction's description has been updated to reflect Java 5 changes.

**Mnemonic:** **putstatic**

**Operation:** Set **static** field in class.

**Opcode:** 179 (0xB3)

**Operands:** ***indexbyte1***, ***indexbyte2***

**Operand Stack:** Before: …, ***value*** >>>
After: …

**Description:** The unsigned ***indexbyte1*** and ***indexbyte2*** construct an index into the current class's runtime constant pool, where the index value is (***indexbyte1***<<8)|***indexbyte2***. The runtime constant pool item at that index must be a symbolic reference to a field, 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 resolves.

On successful resolution of the field, the class or interface that declares the resolved field initializes (unless already initialized).

***value***'s type must be compatible with the referenced field's descriptor. If the field descriptor's type is **boolean**, **byte**, **char**, **int**, or **short**, ***value*** must be an **int**. If the field descriptor's type is **double**, **float**, or **long**, ***value*** must be a **double**, **float**, or **long**, respectively. If the field descriptor's type is a reference type, ***value*** must be of a type that is assignment compatible with the field descriptor type. If the field is **final**, it should be declared in the **<clinit>** method of the current class. Otherwise, an **IllegalAccessError** throws.

***value*** pops from the operand stack and undergoes value set conversion, resulting in ***value'***. The class field sets 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 of the JVM specification can be thrown.

This instruction throws an **IncompatibleClassChangeError** if the resolved field is not a class (**static**) field or an interface field.

This instruction throws an **IllegalAccessError** if the field is **final** but not declared in the **<clinit>** method of the current class.

**Runtime Exceptions:** If execution of this instruction causes initialization of the referenced class or interface, the instruction may throw an **Error** as detailed in section 2.17.5 of the JVM specification.

**Notes:** This 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 initializes.

This instruction's description has been updated to reflect Java 5 changes.

**Mnemonic:** **ret**

**Operation:** Return from subroutine.

**Opcode:** 169 (0xA9)

**Operands:** ***index***

**Operand Stack:** No change

**Description:** ***index*** is an unsigned byte between 0 and 255 (inclusive). The local variable at ***index*** in the current stack frame must contain a value of **returnAddress** type. The contents of the local variable write to the JVM's **pc** register, and execution continues there.

**Notes:** This instruction is used with **jsr** and **jsr_w** to implement Java language **finally** clauses. **jsr** / **jsr_w** pushes the address to the operand stack and this instruction retrieves that address from a local variable. That asymmetry is intentional.

Do not confuse this instruction with **return**. A **return** instruction returns control from a method to its invoker, without passing any value back to the invoker.

This instruction's opcode can be used with the **wide** instruction's opcode to access a local variable using a two-byte unsigned index.

**Mnemonic:** **return**

**Operation:** Return **void** from method.

**Opcode:** 177 (0xB1)

**Operands:** None

**Operand Stack:** Before: … >>>
After: [empty]

**Description:** The current method must have the **void** return type. If the method is **synchronized**, the monitor acquired or reentered on the method's invocation releases or exits (respectively) as if by executing **monitorexit**. If no exception is thrown, any other values on the current method's operand stack discard.

The interpreter returns control to the current method's invoker, reinstating the invoker's stack frame as the current stack frame.

**Runtime Exceptions:** This instruction throws an **IllegalMonitorStateException** 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.

This instruction also throws an **IllegalMonitorStateException** if the JVM implementation enforces the rules on structured lock usage, and the number of lock operations performed by a thread on a lock does not match the number of unlock operations by that thread on that lock (whether the method invocation completes normally or abruptly).

**Mnemonic:** **saload**

**Operation:** Load **short** from array.

**Opcode:** 53 (0x35)

**Operands:** None

**Operand Stack:** Before: …, ***arrayref***, ***index*** >>>
After: …, ***value***

**Description:** ***arrayref*** must be of **reference** type and must refer to an array whose components are of **short** type. Furthermore, ***index*** must be of **int** type. Both ***arrayref*** and ***index*** pop from the operand stack. The array component at ***index*** retrieves and sign-extends to an **int** ***value***, which pushes to the operand stack.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***arrayref*** is **null**.

This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array.

**Mnemonic:** **sastore**

**Operation:** Store into **short** array.

**Opcode:** 86 (0x56)

**Operands:** None

**Operand Stack:** Before: …, ***arrayref***, ***index***, ***value*** >>>
After: …

**Description:** ***arrayref*** must be of **reference** type and must refer to an array whose components are of **short** type. Furthermore, ***index*** and ***value*** must both be of **int** type. ***arrayref***, ***index***, and ***value*** pop from the operand stack. The **int** ***value*** truncates to a **short**, which stores as the array component at ***index***.

**Runtime Exceptions:** This instruction throws a **NullPointerException** if ***arrayref*** is **null**.

This instruction throws an **ArrayIndexOutOfBoundsException** if ***index*** is not within the bounds of the ***arrayref***-referenced array.

**Mnemonic:** **sipush**

**Operation:** Push **short**.

**Opcode:** 17 (0x11)

**Operands:** ***byte1***, ***byte2***

**Operand Stack:** Before: … >>>
After: …, ***value***

**Description:** Assemble unsigned ***byte1*** and ***byte2*** into an intermediate **short**, via (***byte1***<<8)|***byte2***. Sign-extend the intermediate **short** to an **int** ***value***, and then push that ***value*** to the operand stack.

**Mnemonic:** **swap**

**Operation:** Swap the top two operand stack values.

**Opcode:** 95 (0x5F)

**Operands:** None

**Operand Stack:** Before: …, ***value2***, ***value1***>>>
After: …, ***value1***, ***value2***

**Description:** Swap the top two values on the operand stack.

***value1*** and ***value2*** must not be of type **double** or **long**.

**Notes:** The JVM doesn't provide an instruction that swaps operands of type **double** or **long**.

**Mnemonic:** **tableswitch**

**Operation:** Access jump table by index and jump.

**Opcode:** 170 (0xAA)

**Operands:** ***<0-3 byte pad>***, ***defaultbyte1***, ***defaultbyte2***, ***defaultbyte3***, ***defaultbyte4***, ***lowbyte1***, ***lowbyte2***, ***lowbyte3***, ***lowbyte4***, ***highbyte1***, ***highbyte2***, ***highbyte3***, ***highbyte4***, ***jump offsets…***

**Operand Stack:** Before: …, ***index*** >>>
After: …

**Description:** This instruction is variable-length. Immediately after its opcode, between 0 and 3 null bytes (zero bytes, not the null object) insert 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 constructs as (***byte1***<<24)|(***byte2***<<16)|(***byte3***<<8)|***byte4***.

***index*** must be of **int** type and pops from the operand stack. If ***index*** is less than ***low*** or greater than ***high***, a target address calculates by adding ***default*** to the address of this instruction's opcode. Otherwise, the offset at position ***index***-***low*** of the jump table extracts. The target address calculates by adding that offset to this instruction's opcode address. Execution continues at the target address.

The target address that can be calculated from each jump table offset, as well as the ones 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 this instruction guarantees 4-byte alignment of those operands if and only if the method containing this instruction starts on a 4-byte boundary.

**Mnemonic:** **wide**

**Operation:** Extend local variable index by additional bytes.

**Opcode:** 196 (0xC4)

**Operands:** ***<opcode>***, ***indexbyte1***, ***indexbyte2***

where ***<opcode>*** is one of **aload**, **astore**, **dload**, **dstore**, **fload**, **fstore**, **iload**, **istore**, **lload**, **lstore**, or **ret**.

or

**iinc**, ***indexbyte1***, ***indexbyte2***, ***constbyte1***, ***constbyte2***

**Operand Stack:** Same as modified instruction

**Description:** **wide** modifies the behavior of another instruction. It takes one of two operands formats, depending on the instruction being modified. The first format modifies one of instructions **aload**, **astore**, **dload**, **dstore**, **fload**, **fstore**, **iload**, **istore**, **lload**, **lstore**, or **ret**. The second format applies only to the **iinc** instruction.

In either case, the **wide** opcode is followed (in the compiled code) by the opcode of the instruction **wide** modifies. In either format, unsigned bytes ***indexbyte1*** and ***indexbyte2*** follow the modified opcode and assemble into a 16-bit unsigned index to a local variable in the current stack frame, where the value of the index is (***indexbyte1***<<8)|***indexbyte2***. The calculated index must be an index into the current stack frame's local variable array. Where **wide** modifies a **dload**, **dstore**, **lload**, or **lstore** instruction, the index following the calculated index (index+1) must also be an index into the local variable array. In the second format, unsigned bytes ***constbyte1*** and ***constbyte2*** follow ***indexbyte1*** and ***indexbyte2*** in the code stream. Those bytes assemble into a signed 16-bit constant, where the constant is (***constbyte1***<<8)|***constbyte2***.

The widened bytecode operates as normal, except for the use of a wider index and, in the case of the second operands format, the larger increment range.

**Notes:** Although **wide** is said to modify the behavior of another instruction, **wide** 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 **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.