aaload ***aaload***

Operation Load `reference` from array

Format

aaload

Forms *aaload* = 50 (0x32)

Operand Stack ..., *arrayref*, *index* →

..., *value*

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

Run-time Exceptions If *arrayref* is `null`, *aaload* throws a `NullPointerException`.

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

aastore ***aastore***

Operation Store into `reference` array

Format

aastore

Forms *aastore* = 83 (0x53)

Operand Stack ..., *arrayref*, *index*, *value* →

...

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

If *value* is `null`, then *value* is stored as the component of the array at *index*.

Otherwise, *value* is non-`null`. If the type of *value* is assignment compatible with the type of the components of the array referenced by *arrayref*, then *value* is stored as the component of the array at *index*.

The following rules are used to determine whether a *value* that is not `null` is assignment compatible with the array component type. If *S* is the type of the object referred to by *value*, and *T* is the reference type of the array components, then *aastore* determines whether assignment is compatible as follows:

- If `S` is a class type, then:
  - If `T` is a class type, then `S` must be the same class as `T`, or `S` must be a subclass of `T`;
  - If `T` is an interface type, then `S` must implement interface `T`.
- If `S` is an array type `SC[]`, that is, an array of components of type `SC`, then:
  - If `T` is a class type, then `T` must be `Object`.
  - If `T` is an interface type, then `T` must be one of the interfaces implemented by arrays (JLS §4.10.3).
  - If `T` is an array type `TC[]`, that is, an array of components of type `TC`, then one of the following must be true:
    - › `TC` and `SC` are the same primitive type.
    - › `TC` and `SC` are reference types, and type `SC` is assignable to `TC` by these run-time rules.

Run-time Exceptions

If *arrayref* is `null`, *aastore* throws a `NullPointerException`.

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

Otherwise, if *arrayref* is not `null` and the actual type of the non-`null` *value* is not assignment compatible with the actual type of the components of the array, *aastore* throws an `ArrayStoreException`.

aconst_null *aconst_null*

Operation Push `null`

Format

aconst_null

Forms *aconst_null* = 1 (0x1)

Operand Stack ... →

..., `null`

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

Notes The Java Virtual Machine does not mandate a concrete value for `null`.

aload *aload*

Operation Load `reference` from local variable

Format

aload
index

Forms *aload* = 25 (0x19)

Operand Stack ... →

..., *objectref*

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `reference`. The *objectref* in the local variable at *index* is pushed onto the operand stack.

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

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

aload_<n> *aload_<n>*

Operation Load `reference` from local variable

Format

aload_<n>

Forms

aload_0 = 42 (0x2a)

aload_1 = 43 (0x2b)

aload_2 = 44 (0x2c)

aload_3 = 45 (0x2d)

Operand Stack

... →

..., *objectref*

Description The <*n*> must be an index into the local variable array of the current frame (§2.6). The local variable at <*n*> must contain a `reference`. The *objectref* in the local variable at <*n*> is pushed onto the operand stack.

Notes An *aload_<n>* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding *astore_<n>* instruction (§*astore_<n>*) is intentional.

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

anewarray *anewarray*

Operation Create new array of `reference`

Format

anewarray
indexbyte1
indexbyte2

Forms anewarray = 189 (0xbd)

Operand Stack

..., *count* →

..., *arrayref*

Description The *count* must be of type `int`. It is popped off the operand stack. The *count* represents the number of components of the array to be created. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1). A new array with components of that type, of length *count*, is allocated from the garbage-collected heap, and a `reference` *arrayref* to this new array object is pushed onto the operand stack. All components of the new array are initialized to `null`, the default value for `reference` types (§2.4).

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

Run-time Exceptions Otherwise, if *count* is less than zero, the *anewarray* instruction throws a `NegativeArraySizeException`.

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

areturn *areturn*

Operation Return `reference` from method

Format

areturn

Forms *areturn* = 176 (0xb0)

Operand Stack ..., *objectref* →

[empty]

Description The *objectref* must be of type `reference` and must refer to an object of a type that is assignment compatible (JLS §5.2) with the type represented by the return descriptor (§4.3.3) of the current method. If the current method is a `synchronized` method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread. If no exception is thrown, *objectref* is popped from the operand stack of the current frame (§2.6) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

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

Run-time Exceptions If the Java Virtual Machine implementation does not enforce the rules on structured locking described in §2.11.10, then if the current method is a `synchronized` method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, *areturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a `synchronized` method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the first of those rules is violated during invocation of the current method, then *areturn* throws an `IllegalMonitorStateException`.

arraylength *arraylength*

Operation Get length of array

Format

arraylength

Forms *arraylength* = 190 (0xbe)

Operand Stack ..., *arrayref* →

..., *length*

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

Run-time Exceptions If the *arrayref* is `null`, the *arraylength* instruction throws a `NullPointerException`.

astore *astore*

Operation Store `reference` into local variable

Format

astore
index

Forms *astore* = 58 (0x3a)

Operand Stack ..., *objectref* →

...

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at *index* is set to *objectref*.

Notes The *astore* instruction is used with an *objectref* of type `returnAddress` when implementing the `finally` clause of the Java programming language (§3.13).

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

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

astore_<n> *astore_<n>*

Operation Store `reference` into local variable

Format

astore_<n>

Forms *astore_0* = 75 (0x4b)

astore_1 = 76 (0x4c)

astore_2 = 77 (0x4d)

astore_3 = 78 (0x4e)

Operand Stack ..., *objectref* →

...

Description The <*n*> must be an index into the local variable array of the current frame (§2.6). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *objectref*.

Notes An *astore_<n>* instruction is used with an *objectref* of type `returnAddress` when implementing the `finally` clauses of the Java programming language (§3.13).

An *aload_<n>* instruction (§*aload_<n>*) cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding *astore_<n>* instruction is intentional.

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

athrow *athrow*

Operation Throw exception or error

Format

athrow

Forms *athrow* = 191 (0xbf)

Operand Stack ..., *objectref* →

objectref

Description The *objectref* must be of type `reference` and must refer to an object that is an instance of class `Throwable` or of a subclass of `Throwable`. It is popped from the operand stack. The *objectref* is then thrown by searching the current method (§2.6) for the first exception handler that matches the class of *objectref*, as given by the algorithm in §2.10.

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

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

Run-time Exceptions If *objectref* is `null`, *athrow* throws a `NullPointerException` instead of *objectref*.

Otherwise, if the Java Virtual Machine implementation does not enforce the rules on structured locking described in §2.11.10, then if the method of the current frame is a `synchronized` method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, *athrow* throws an `IllegalMonitorStateException` instead of the object previously being thrown. This can happen, for example, if an abruptly completing `synchronized` method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the first of those rules is violated during invocation of the current method, then *athrow* throws an `IllegalMonitorStateException` instead of the object previously being thrown.

Notes

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

baload *baload*

Operation Load `byte` or `boolean` from array

Format

baload

Forms *baload* = 51 (0x33)

Operand Stack

..., *arrayref*, *index* →

..., *value*

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

Run-time Exceptions If *arrayref* is `null`, *baload* throws a `NullPointerException`.

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

Notes The *baload* instruction is used to load values from both `byte` and `boolean` arrays. In Oracle's Java Virtual Machine implementation, `boolean` arrays - that is, arrays of type `T_BOOLEAN` (§2.2, §*newarray*) - are implemented as arrays of 8-bit values. Other implementations may implement packed `boolean` arrays; the *baload* instruction of such implementations must be used to access those arrays.

bastore *bastore*

Operation Store into `byte` or `boolean` array

Format

bastore

Forms *bastore* = 84 (0x54)

Operand Stack ..., *arrayref*, *index*, *value* →

...

Description The *arrayref* must be of type `reference` and must refer to an array whose components are of type `byte` or of type `boolean`. The *index* and the *value* must both be of type `int`. The *arrayref*, *index*, and *value* are popped from the operand stack.

If the *arrayref* refers to an array whose components are of type `byte`, then the `int` *value* is truncated to a `byte` and stored as the component of the array indexed by *index*.

If the *arrayref* refers to an array whose components are of type `boolean`, then the `int` *value* is narrowed by taking the bitwise AND of *value* and 1; the result is stored as the component of the array indexed by *index*.

Run-time Exceptions If *arrayref* is `null`, *bastore* throws a `NullPointerException`.

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

Notes The *bastore* instruction is used to store values into both `byte` and `boolean` arrays. In Oracle's Java Virtual Machine implementation, `boolean` arrays - that is, arrays of type `T_BOOLEAN` (§2.2, §*newarray*) - are implemented as arrays of 8-bit values. Other implementations may implement packed `boolean` arrays; in such implementations the *bastore* instruction must be able to store `boolean` values into packed `boolean` arrays as well as `byte` values into `byte` arrays.

bipush ***bipush***

Operation Push `byte`

Format

bipush
byte

Forms *bipush* = 16 (0x10)

Operand Stack ... →

..., *value*

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

caload *caload*

Operation Load `char` from array

Format

caload

Forms *caload* = 52 (0x34)

Operand Stack ..., *arrayref*, *index* →

..., *value*

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

Run-time Exceptions If *arrayref* is `null`, *caload* throws a `NullPointerException`.

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

castore ***castore***

Operation Store into `char` array

Format

castore

Forms *castore* = 85 (0x55)

Operand Stack ..., *arrayref*, *index*, *value* →

...

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

Run-time Exceptions If *arrayref* is `null`, *castore* throws a `NullPointerException`.

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

checkcast *checkcast*

Operation Check whether object is of given type

Format

checkcast
indexbyte1
indexbyte2

Forms *checkcast* = 192 (0xc0)

Operand Stack ..., *objectref* →

..., *objectref*

Description The *objectref* must be of type `reference`. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a class, array, or interface type.

If *objectref* is `null`, then the operand stack is unchanged.

Otherwise, the named class, array, or interface type is resolved (§5.4.3.1). If *objectref* can be cast to the resolved class, array, or interface type, the operand stack is unchanged; otherwise, the *checkcast* instruction throws a `ClassCastException`.

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

- If `S` is a class type, then:
  - If `T` is a class type, then `S` must be the same class as `T`, or `S` must be a subclass of `T`;
  - If `T` is an interface type, then `S` must implement interface `T`.
- If `S` is an array type `SC[]`, that is, an array of components of type `SC`, then:
  - If `T` is a class type, then `T` must be `Object`.
  - If `T` is an interface type, then `T` must be one of the interfaces implemented by arrays (JLS §4.10.3).
  - If `T` is an array type `TC[]`, that is, an array of components of type `TC`, then one of the following must be true:
    - › `TC` and `SC` are the same primitive type.
    - › `TC` and `SC` are reference types, and type `SC` can be cast to `TC` by recursive application of these rules.

Linking Exceptions

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

Run-time Exception

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

Notes

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

d2f *d2f*

Operation Convert `double` to `float`

Format

d2f

Forms *d2f* = 144 (0x90)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and converted to a `float` *result* using the round to nearest rounding policy (§2.8). The *result* is pushed onto the operand stack.

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

Notes The *d2f* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value* and may also lose precision.

d2i *d2i*

Operation Convert `double` to `int`

Format

d2i

Forms *d2i* = 142 (0x8e)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and converted to an `int` *result*. The *result* is pushed onto the operand stack:

- If the *value* is NaN, the result of the conversion is an `int` 0.
- Otherwise, if the *value* is not an infinity, it is rounded to an integer value *V* using the round toward zero rounding policy (§2.8). If this integer value *V* can be represented as an `int`, then the result is the `int` value *V*.
- Otherwise, either the *value* must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type `int`, or the *value* must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type `int`.

Notes The *d2i* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value* and may also lose precision.

d2l *d2l*

Operation Convert `double` to `long`

Format

d2l

Forms *d2l* = 143 (0x8f)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack and converted to a `long`. The *result* is pushed onto the operand stack:

- If the *value* is NaN, the result of the conversion is a `long` 0.
- Otherwise, if the *value* is not an infinity, it is rounded to an integer value *V* using the round toward zero rounding policy (§2.8). If this integer value *V* can be represented as a `long`, then the result is the `long` value *V*.
- Otherwise, either the *value* must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type `long`, or the *value* must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type `long`.

Notes The *d2l* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value* and may also lose precision.

dadd *dadd*

Operation Add `double`

Format

dadd

Forms *dadd* = 99 (0x63)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

The result of a *dadd* instruction is governed by the rules of IEEE 754 arithmetic:

- If either *value1* or *value2* is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using the round to nearest rounding policy (§2.8). If the magnitude is too large to represent as a `double`,

we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `double`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java Virtual Machine requires support of gradual underflow. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a *dadd* instruction never throws a run-time exception.

daload *daload*

Operation Load `double` from array

Format

daload

Forms *daload* = 49 (0x31)

Operand Stack ..., *arrayref*, *index* →

..., *value*

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

Run-time Exceptions If *arrayref* is `null`, *daload* throws a `NullPointerException`.

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

dastore *dastore*

Operation Store into `double` array

Format

dastore

Forms *dastore* = 82 (0x52)

Operand Stack ..., *arrayref*, *index*, *value* →

...

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

Run-time Exceptions If *arrayref* is `null`, *dastore* throws a `NullPointerException`.

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

dcmp<op> *dcmp<op>*

Operation Compare `double`

Format

dcmp<op>

Forms *dcmpg* = 152 (0x98)

dcmpl = 151 (0x97)

Operand Stack ..., *value1*, *value2* →

..., *result*

Description Both *value1* and *value2* must be of type `double`. The values are popped from the operand stack and a floating-point comparison is performed:

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

Floating-point comparison is performed in accordance with IEEE 754. All values other than NaN are ordered, with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.

Notes The *dcmpg* and *dcmpl* instructions differ only in their treatment of a comparison involving NaN. NaN is unordered, so any `double` comparison fails if either or both of its operands are NaN. With both *dcmpg* and *dcmpl* available, any `double` comparison may be compiled to push the same *result* onto the operand stack whether the comparison fails on non-NaN values or fails because it encountered a NaN. For more information, see §3.5.

dconst_<d> *dconst_<d>*

Operation Push `double`

Format

dconst_<d>

Forms *dconst_0* = 14 (0xe)

dconst_1 = 15 (0xf)

Operand Stack ... →

..., *<d>*

Description Push the `double` constant *<d>* (0.0 or 1.0) onto the operand stack.

ddiv *ddiv*

Operation Divide `double`

Format

ddiv

Forms *ddiv* = 111 (0x6f)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

The result of a *ddiv* instruction is governed by the rules of IEEE 754 arithmetic:

- If either *value1* or *value2* is NaN, the result is NaN.
- If neither *value1* nor *value2* is NaN, the sign of the result is positive if both values have the same sign, negative if the values have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- Division of a finite value by an infinity results in a signed zero, with the sign-producing rule just given.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero, with the sign-producing rule just given.
- Division of a nonzero finite value by a zero results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the quotient is computed and rounded to the nearest `double` using the round to nearest rounding policy (§2.8). If the magnitude is too large to represent as a `double`,

we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `double`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java Virtual Machine requires support of gradual underflow. Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of a *ddiv* instruction never throws a run-time exception.

dload *dload*

Operation Load `double` from local variable

Format

dload
index

Forms *dload* = 24 (0x18)

Operand Stack ... →

..., *value*

Description The *index* is an unsigned byte. Both *index* and *index*+1 must be indices into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `double`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes The *dload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

dload_<n> *dload_<n>*

Operation Load `double` from local variable

Format

dload_<n>

Forms

dload_0 = 38 (0x26)

dload_1 = 39 (0x27)

dload_2 = 40 (0x28)

dload_3 = 41 (0x29)

Operand Stack

... →

..., *value*

Description Both <*n*> and <*n*>+1 must be indices into the local variable array of the current frame (§2.6). The local variable at <*n*> must contain a `double`. The *value* of the local variable at <*n*> is pushed onto the operand stack.

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

dmul *dmul*

Operation Multiply `double`

Format

dmul

Forms *dmul* = 107 (0x6b)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

The result of a *dmul* instruction is governed by the rules of IEEE 754 arithmetic:

- If either *value1* or *value2* is NaN, the result is NaN.
- If neither *value1* nor *value2* is NaN, the sign of the result is positive if both values have the same sign and negative if the values have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither an infinity nor NaN is involved, the product is computed and rounded to the nearest representable value using the round to nearest rounding policy (§2.8). If the magnitude is too large to represent as a `double`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `double`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java Virtual Machine requires support of gradual underflow. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a *dmul* instruction never throws a run-time exception.

dneg *dneg*

Operation Negate `double`

Format

dneg

Forms *dneg* = 119 (0x77)

Operand Stack ..., *value* →

..., *result*

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

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

Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).

  The Java Virtual Machine has not adopted the stronger requirement from the 2019 version of the IEEE 754 Standard that negation inverts the sign bit for all inputs, including NaN.

- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.

drem *drem*

Operation Remainder `double`

Format

drem

Forms *drem* = 115 (0x73)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

The result of a *drem* instruction is not the same as the result of the remainder operation defined by IEEE 754, due to the choice of rounding policy in the Java Virtual Machine (§2.8). The IEEE 754 remainder operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead, the Java Virtual Machine defines *drem* to behave in a manner analogous to that of the integer remainder instructions *irem* and *lrem*, with an implied division using the round toward zero rounding policy; this may be compared with the C library function `fmod`.

The result of a *drem* instruction is governed by the following rules, which match IEEE 754 arithmetic except for how the implied division is computed:

- If either *value1* or *value2* is NaN, the result is NaN.
- If neither *value1* nor *value2* is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity or the divisor is a zero or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the dividend.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder *result* from a dividend *value1* and a divisor *value2* is defined by the mathematical relation *result* = *value1* - (*value2* * *q*), where *q* is an integer that is negative only if *value1* / *value2* is negative, and positive only if *value1* / *value2* is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of *value1* and *value2*.

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

Notes The IEEE 754 remainder operation may be computed by the library routine `Math.IEEEremainder` or `StrictMath.IEEEremainder`.

dreturn

Operation Return `double` from method

Format

dreturn

Forms *dreturn* = 175 (0xaf)

Operand Stack

..., *value* →

[empty]

Description

The current method must have return type `double`. The *value* must be of type `double`. If the current method is a `synchronized` method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread. If no exception is thrown, *value* is popped from the operand stack of the current frame (§2.6) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

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

Run-time Exceptions

If the Java Virtual Machine implementation does not enforce the rules on structured locking described in §2.11.10, then if the current method is a `synchronized` method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, *dreturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a `synchronized` method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the first of those rules is violated during invocation of the current method, then *dreturn* throws an `IllegalMonitorStateException`.

dstore *dstore*

Operation Store `double` into local variable

Format

dstore
index

Forms *dstore* = 57 (0x39)

Operand Stack ..., *value* →

...

Description The *index* is an unsigned byte. Both *index* and *index*+1 must be indices into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack. The local variables at *index* and *index*+1 are set to *value*.

Notes The *dstore* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

dstore_<n>   *dstore_<n>*

Operation Store `double` into local variable

Format

dstore_<n>

Forms

dstore_0 = 71 (0x47)

dstore_1 = 72 (0x48)

dstore_2 = 73 (0x49)

dstore_3 = 74 (0x4a)

Operand Stack

..., *value* →

...

Description Both <*n*> and <*n*>+1 must be indices into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `double`. It is popped from the operand stack. The local variables at <*n*> and <*n*>+1 are set to *value*.

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

dsub *dsub*

Operation Subtract `double`

Format

dsub

Forms *dsub* = 103 (0x67)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

The Java Virtual Machine requires support of gradual underflow. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a *dsub* instruction never throws a run-time exception.

dup ***dup***

Operation Duplicate the top operand stack value

Format

dup

Forms *dup* = 89 (0x59)

Operand Stack ..., *value* →

..., *value*, *value*

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

The *dup* instruction must not be used unless *value* is a value of a category 1 computational type (§2.11.1).

dup_x1 *dup_x1*

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

Format

dup_x1

Forms *dup_x1* = 90 (0x5a)

Operand Stack ..., *value2*, *value1* →

..., *value1*, *value2*, *value1*

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

The *dup_x1* instruction must not be used unless both *value1* and *value2* are values of a category 1 computational type (§2.11.1).

dup_x2 *dup_x2*

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

Format

dup_x2

Forms *dup_x2* = 91 (0x5b)

Operand Stack

Form 1:

..., *value3*, *value2*, *value1* →

..., *value1*, *value3*, *value2*, *value1*

where *value1*, *value2*, and *value3* are all values of a category 1 computational type (§2.11.1).

Form 2:

..., *value2*, *value1* →

..., *value1*, *value2*, *value1*

where *value1* is a value of a category 1 computational type and *value2* is a value of a category 2 computational type (§2.11.1).

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

dup2 *dup2*

Operation Duplicate the top one or two operand stack values

Format

dup2

Forms *dup2* = 92 (0x5c)

Operand Stack

Form 1:

..., *value2*, *value1* →

..., *value2*, *value1*, *value2*, *value1*

where both *value1* and *value2* are values of a category 1 computational type (§2.11.1).

Form 2:

..., *value* →

..., *value*, *value*

where *value* is a value of a category 2 computational type (§2.11.1).

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

dup2_x1 ***dup2_x1***

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

Format

dup2_x1

Forms *dup2_x1* = 93 (0x5d)

Operand Stack Form 1:

..., *value3*, *value2*, *value1* →

..., *value2*, *value1*, *value3*, *value2*, *value1*

where *value1*, *value2*, and *value3* are all values of a category 1 computational type (§2.11.1).

Form 2:

..., *value2*, *value1* →

..., *value1*, *value2*, *value1*

where *value1* is a value of a category 2 computational type and *value2* is a value of a category 1 computational type (§2.11.1).

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

dup2_x2 *dup2_x2*

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

Format

dup2_x2

Forms *dup2_x2* = 94 (0x5e)

Operand Stack

Form 1:

..., *value4*, *value3*, *value2*, *value1* →

..., *value2*, *value1*, *value4*, *value3*, *value2*, *value1*

where *value1*, *value2*, *value3*, and *value4* are all values of a category 1 computational type (§2.11.1).

Form 2:

..., *value3*, *value2*, *value1* →

..., *value1*, *value3*, *value2*, *value1*

where *value1* is a value of a category 2 computational type and *value2* and *value3* are both values of a category 1 computational type (§2.11.1).

Form 3:

..., *value3*, *value2*, *value1* →

..., *value2*, *value1*, *value3*, *value2*, *value1*

where *value1* and *value2* are both values of a category 1 computational type and *value3* is a value of a category 2 computational type (§2.11.1).

Form 4:

..., *value2*, *value1* →

..., *value1*, *value2*, *value1*

where *value1* and *value2* are both values of a category 2 computational type (§2.11.1).

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

f2d *f2d*

Operation Convert `float` to `double`

Format

f2d

Forms *f2d* = 141 (0x8d)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `float`. It is popped from the operand stack and converted to a `double` *result*. The *result* is pushed onto the operand stack.

Notes The *f2d* instruction performs a widening primitive conversion (JLS §5.1.2).

f2i *f2i*

Operation Convert `float` to `int`

Format

f2i

Forms *f2i* = 139 (0x8b)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `float`. It is popped from the operand stack and converted to an `int` *result*. The *result* is pushed onto the operand stack:

- If the *value* is NaN, the *result* of the conversion is an `int` 0.
- Otherwise, if the *value* is not an infinity, it is rounded to an integer value *V* using the round toward zero rounding policy (§2.8). If this integer value *V* can be represented as an `int`, then the *result* is the `int` value *V*.
- Otherwise, either the *value* must be too small (a negative value of large magnitude or negative infinity), and the *result* is the smallest representable value of type `int`, or the *value* must be too large (a positive value of large magnitude or positive infinity), and the *result* is the largest representable value of type `int`.

Notes The *f2i* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value* and may also lose precision.

f2l

Operation Convert `float` to `long`

Format

f2l

Forms *f2l* = 140 (0x8c)

Operand Stack

..., *value* →

..., *result*

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

- If the *value* is NaN, the result of the conversion is a `long` 0.
- Otherwise, if the *value* is not an infinity, it is rounded to an integer value *V* using the round toward zero rounding policy (§2.8). If this integer value *V* can be represented as a `long`, then the *result* is the `long` value *V*.
- Otherwise, either the *value* must be too small (a negative value of large magnitude or negative infinity), and the *result* is the smallest representable value of type `long`, or the *value* must be too large (a positive value of large magnitude or positive infinity), and the *result* is the largest representable value of type `long`.

Notes The *f2l* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value* and may also lose precision.

fadd *fadd*

Operation Add `float`

Format

fadd

Forms *fadd* = 98 (0x62)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

The result of an *fadd* instruction is governed by the rules of IEEE 754 arithmetic:

- If either *value1* or *value2* is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using the round to nearest rounding policy (§2.8). If the magnitude is too large to represent as a `float`,

we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `float`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java Virtual Machine requires support of gradual underflow. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an *fadd* instruction never throws a run-time exception.

faload ***faload***

Operation Load `float` from array

Format

faload

Forms *faload* = 48 (0x30)

Operand Stack ..., *arrayref*, *index* →

..., *value*

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

Run-time Exceptions If *arrayref* is `null`, *faload* throws a `NullPointerException`.

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

fastore *fastore*

Operation Store into `float` array

Format

fastore

Forms *fastore* = 81 (0x51)

Operand Stack ..., *arrayref*, *index*, *value* →

...

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

Run-time Exceptions If *arrayref* is `null`, *fastore* throws a `NullPointerException`.

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

fcmp<op> ***fcmp<op>***

Operation Compare `float`

Format

fcmp<op>

Forms *fcmpg* = 150 (0x96)

fcmpl = 149 (0x95)

Operand Stack ..., *value1*, *value2* →

..., *result*

Description Both *value1* and *value2* must be of type `float`. The values are popped from the operand stack and a floating-point comparison is performed:

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

Floating-point comparison is performed in accordance with IEEE 754. All values other than NaN are ordered, with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.

Notes The *fcmpg* and *fcmpl* instructions differ only in their treatment of a comparison involving NaN. NaN is unordered, so any `float` comparison fails if either or both of its operands are NaN. With both *fcmpg* and *fcmpl* available, any `float` comparison may be compiled to push the same *result* onto the operand stack whether the comparison fails on non-NaN values or fails because it encountered a NaN. For more information, see §3.5.

fconst_<f> *fconst_<f>*

Operation Push `float`

Format

fconst_<f>

Forms *fconst_0* = 11 (0xb)

fconst_1 = 12 (0xc)

fconst_2 = 13 (0xd)

Operand Stack ... →

..., <*f*>

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

fdiv

Operation Divide `float`

Format

fdiv

Forms *fdiv* = 110 (0x6e)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

The result of an *fdiv* instruction is governed by the rules of IEEE 754 arithmetic:

- If either *value1* or *value2* is NaN, the result is NaN.
- If neither *value1* nor *value2* is NaN, the sign of the result is positive if both values have the same sign, negative if the values have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- Division of a finite value by an infinity results in a signed zero, with the sign-producing rule just given.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero, with the sign-producing rule just given.
- Division of a nonzero finite value by a zero results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the quotient is computed and rounded to the nearest `float` using the round to nearest rounding policy (§2.8). If the magnitude is too large to represent as a `float`, we say the

operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `float`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java Virtual Machine requires support of gradual underflow. Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of an *fdiv* instruction never throws a run-time exception.

fload *fload*

Operation Load `float` from local variable

Format

fload
index

Forms *fload* = 23 (0x17)

Operand Stack ... →

..., *value*

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `float`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes The *fload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

fload_<n> *fload_<n>*

Operation Load `float` from local variable

Format

fload_<n>

Forms *fload_0* = 34 (0x22)

fload_1 = 35 (0x23)

fload_2 = 36 (0x24)

fload_3 = 37 (0x25)

Operand Stack ... →

..., *value*

Description The <*n*> must be an index into the local variable array of the current frame (§2.6). The local variable at <*n*> must contain a `float`. The *value* of the local variable at <*n*> is pushed onto the operand stack.

Notes Each of the *fload_<n>* instructions is the same as *fload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

fmul *fmul*

Operation Multiply `float`

Format

fmul

Forms *fmul* = 106 (0x6a)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

The result of an *fmul* instruction is governed by the rules of IEEE 754 arithmetic:

- If either *value1* or *value2* is NaN, the result is NaN.
- If neither *value1* nor *value2* is NaN, the sign of the result is positive if both values have the same sign, and negative if the values have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither an infinity nor NaN is involved, the product is computed and rounded to the nearest representable value using the round to nearest rounding policy (§2.8). If the magnitude is too large to represent as a `float`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `float`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java Virtual Machine requires support of gradual underflow. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an *fmul* instruction never throws a run-time exception.

fneg ***fneg***

Operation Negate `float`

Format

fneg

Forms *fneg* = 118 (0x76)

Operand Stack

..., *value* →

..., *result*

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

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

Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).

  The Java Virtual Machine has not adopted the stronger requirement from the 2019 version of the IEEE 754 Standard that negation inverts the sign bit for all inputs, including NaN.

- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.

frem *frem*

Operation Remainder `float`

Format

frem

Forms *frem* = 114 (0x72)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

The result of an *frem* instruction is not the same as the result of the remainder operation defined by IEEE 754, due to the choice of rounding policy in the Java Virtual Machine (§2.8). The IEEE 754 remainder operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead, the Java Virtual Machine defines *frem* to behave in a manner analogous to that of the integer remainder instructions *irem* and *lrem*, with an implied division using the round toward zero rounding policy; this may be compared with the C library function `fmod`.

The result of an *frem* instruction is governed by the following rules, which match IEEE 754 arithmetic except for how the implied division is computed:

- If either *value1* or *value2* is NaN, the result is NaN.
- If neither *value1* nor *value2* is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity or the divisor is a zero or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the dividend.
- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder *result* from a dividend *value1* and a divisor *value2* is defined by the mathematical relation *result* = *value1* - (*value2* * *q*), where *q* is an integer that is negative only if *value1* / *value2* is negative, and positive only if *value1* / *value2* is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of *value1* and *value2*.

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

Notes

The IEEE 754 remainder operation may be computed by the library routine `Math.IEEEremainder` or `StrictMath.IEEEremainder`.

freturn *freturn*

Operation Return `float` from method

Format

freturn

Forms *freturn* = 174 (0xae)

Operand Stack ..., *value* →

[empty]

Description The current method must have return type `float`. The *value* must be of type `float`. If the current method is a `synchronized` method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread. If no exception is thrown, *value* is popped from the operand stack of the current frame (§2.6) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

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

Run-time Exceptions If the Java Virtual Machine implementation does not enforce the rules on structured locking described in §2.11.10, then if the current method is a `synchronized` method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, *freturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a `synchronized` method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the first of those rules is violated during invocation of the current method, then *freturn* throws an `IllegalMonitorStateException`.

fstore ***fstore***

Operation Store `float` into local variable

Format

fstore
index

Forms *fstore* = 56 (0x38)

Operand Stack ..., *value* →

...

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `float`. It is popped from the operand stack, and the value of the local variable at *index* is set to *value*.

Notes The *fstore* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

fstore_<n> *fstore_<n>*

Operation Store `float` into local variable

Format

fstore_<n>

Forms *fstore_0* = 67 (0x43)

fstore_1 = 68 (0x44)

fstore_2 = 69 (0x45)

fstore_3 = 70 (0x46)

Operand Stack ..., *value* →

...

Description The <*n*> must be an index into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `float`. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *value*.

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

fsub *fsub*

Operation Subtract `float`

Format

fsub

Forms *fsub* = 102 (0x66)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

The Java Virtual Machine requires support of gradual underflow. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an *fsub* instruction never throws a run-time exception.

getfield

Operation Fetch field from object

Format

getfield
indexbyte1
indexbyte2

Forms *getfield* = 180 (0xb4)

Operand Stack ..., *objectref* →

..., *value*

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.2).

The *objectref*, which must be of type `reference` but not an array type, is popped from the operand stack. The *value* of the referenced field in *objectref* is fetched and pushed onto the operand stack.

Linking Exceptions During resolution of the symbolic reference to the field, any of the errors pertaining to field resolution (§5.4.3.2) can be thrown.

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

Run-time Exception Otherwise, if *objectref* is `null`, the *getfield* instruction throws a `NullPointerException`.

Notes The *getfield* instruction cannot be used to access the `length` field of an array. The *arraylength* instruction (§*arraylength*) is used instead.

getstatic *getstatic*

Operation Get `static` field from class

Format

getstatic
indexbyte1
indexbyte2

Forms *getstatic* = 178 (0xb2)

Operand Stack ..., →

..., *value*

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved (§5.4.3.2).

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

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

Linking Exceptions During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution (§5.4.3.2) can be thrown.

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

Run-time Exception

Otherwise, if execution of this *getstatic* instruction causes initialization of the referenced class or interface, *getstatic* may throw an `Error` as detailed in §5.5.

goto *goto*

Operation Branch always

Format

goto
branchbyte1
branchbyte2

Forms *goto* = 167 (0xa7)

Operand Stack No change

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

goto_w *goto_w*

Operation Branch always (wide index)

Format

goto_w
branchbyte1
branchbyte2
branchbyte3
branchbyte4

Forms *goto_w* = 200 (0xc8)

Operand Stack No change

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

Notes Although the *goto_w* instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes (§4.11). This limit may be raised in a future release of the Java Virtual Machine.

i2b

Operation Convert `int` to `byte`

Format

i2b

Forms *i2b* = 145 (0x91)

Operand Stack

..., *value* →

..., *result*

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

Notes The *i2b* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

i2c *i2c*

Operation Convert `int` to `char`

Format

i2c

Forms *i2c* = 146 (0x92)

Operand Stack ..., *value* →

..., *result*

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

Notes The *i2c* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value*. The *result* (which is always positive) may also not have the same sign as *value*.

i2d ***i2d***

Operation Convert `int` to `double`

Format

i2d

Forms *i2d* = 135 (0x87)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack and converted to a `double` *result*. The *result* is pushed onto the operand stack.

Notes The *i2d* instruction performs a widening primitive conversion (JLS §5.1.2). Because all values of type `int` are exactly representable by type `double`, the conversion is exact.

i2f

Operation Convert `int` to `float`

Format

i2f

Forms *i2f* = 134 (0x86)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack and converted to a `float` *result* using the round to nearest rounding policy (§2.8). The *result* is pushed onto the operand stack.

Notes The *i2f* instruction performs a widening primitive conversion (JLS §5.1.2), but may result in a loss of precision because values of type `float` have only 24 significand bits.

i2l ***i2l***

Operation Convert `int` to `long`

Format

i2l

Forms *i2l* = 133 (0x85)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack and sign-extended to a `long` *result*. The *result* is pushed onto the operand stack.

Notes The *i2l* instruction performs a widening primitive conversion (JLS §5.1.2). Because all values of type `int` are exactly representable by type `long`, the conversion is exact.

i2s *i2s*

Operation Convert `int` to `short`

Format

i2s

Forms *i2s* = 147 (0x93)

Operand Stack ..., *value* →

..., *result*

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

Notes The *i2s* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

iadd *iadd*

Operation Add `int`

Format

iadd

Forms *iadd* = 96 (0x60)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.

iaload *iaload*

Operation Load `int` from array

Format

iaload

Forms *iaload* = 46 (0x2e)

Operand Stack ..., *arrayref*, *index* →

..., *value*

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

Run-time Exceptions If *arrayref* is `null`, *iaload* throws a `NullPointerException`.

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

iand ***iand***

Operation Boolean AND `int`

Format

iand

Forms *iand* = 126 (0x7e)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

iastore *iastore*

Operation Store into `int` array

Format

iastore

Forms *iastore* = 79 (0x4f)

Operand Stack ..., *arrayref*, *index*, *value* →

...

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

Run-time Exceptions If *arrayref* is `null`, *iastore* throws a `NullPointerException`.

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

iconst_<i> *iconst_<i>*

Operation Push `int` constant

Format

iconst_<i>

Forms

iconst_m1 = 2 (0x2)

iconst_0 = 3 (0x3)

iconst_1 = 4 (0x4)

iconst_2 = 5 (0x5)

iconst_3 = 6 (0x6)

iconst_4 = 7 (0x7)

iconst_5 = 8 (0x8)

Operand Stack

... →

..., <*i*>

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

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

idiv *idiv*

Operation Divide `int`

Format

idiv

Forms *idiv* = 108 (0x6c)

Operand Stack ..., *value1*, *value2* →

..., *result*

Description Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is the value of the Java programming language expression *value1* / *value2* (JLS §15.17.2). The *result* is pushed onto the operand stack.

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

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

Run-time Exception If the value of the divisor in an `int` division is 0, *idiv* throws an `ArithmeticException`.

if_acmp<cond> *if_acmp<cond>*

Operation Branch if `reference` comparison succeeds

Format

if_acmp<cond>
branchbyte1
branchbyte2

Forms *if_acmpeq* = 165 (0xa5)

if_acmpne = 166 (0xa6)

Operand Stack ..., *value1*, *value2* →

...

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

- *if_acmpeq* succeeds if and only if *value1* = *value2*
- *if_acmpne* succeeds if and only if *value1* ≠ *value2*

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

Otherwise, if the comparison fails, execution proceeds at the address of the instruction following this *if_acmp<cond>* instruction.

if_icmp<cond> *if_icmp<cond>*

Operation Branch if `int` comparison succeeds

Format

if_icmp<cond>
branchbyte1
branchbyte2

Forms

if_icmpeq = 159 (0x9f)

if_icmpne = 160 (0xa0)

if_icmplt = 161 (0xa1)

if_icmpge = 162 (0xa2)

if_icmpgt = 163 (0xa3)

if_icmple = 164 (0xa4)

Operand Stack

..., *value1*, *value2* →

...

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

- *if_icmpeq* succeeds if and only if *value1* = *value2*
- *if_icmpne* succeeds if and only if *value1* ≠ *value2*
- *if_icmplt* succeeds if and only if *value1* < *value2*
- *if_icmple* succeeds if and only if *value1* ≤ *value2*
- *if_icmpgt* succeeds if and only if *value1* > *value2*
- *if_icmpge* succeeds if and only if *value1* ≥ *value2*

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

Otherwise, execution proceeds at the address of the instruction following this *if_icmp<cond>* instruction.

if<cond> *if<cond>*

Operation Branch if `int` comparison with zero succeeds

Format

if<cond>
branchbyte1
branchbyte2

Forms

ifeq = 153 (0x99)

ifne = 154 (0x9a)

iflt = 155 (0x9b)

ifge = 156 (0x9c)

ifgt = 157 (0x9d)

ifle = 158 (0x9e)

Operand Stack

..., *value* →

...

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

- *ifeq* succeeds if and only if *value* = 0
- *ifne* succeeds if and only if *value* ≠ 0
- *iflt* succeeds if and only if *value* < 0
- *ifle* succeeds if and only if *value* ≤ 0
- *ifgt* succeeds if and only if *value* > 0
- *ifge* succeeds if and only if *value* ≥ 0

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

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

ifnonnull *ifnonnull*

Operation Branch if `reference` not `null`

Format

ifnonnull
branchbyte1
branchbyte2

Forms *ifnonnull* = 199 (0xc7)

Operand Stack ..., *value* →

...

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

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

ifnull *ifnull*

Operation Branch if `reference` is `null`

Format

ifnull
branchbyte1
branchbyte2

Forms *ifnull* = 198 (0xc6)

Operand Stack ..., *value* →

...

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

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

iinc *iinc*

Operation Increment local variable by constant

Format

iinc
index
const

Forms *iinc* = 132 (0x84)

Operand Stack No change

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The *const* is an immediate signed byte. The local variable at *index* must contain an `int`. The value *const* is first sign-extended to an `int`, and then the local variable at *index* is incremented by that amount.

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

iload *iload*

Operation Load `int` from local variable

Format

iload
index

Forms *iload* = 21 (0x15)

Operand Stack ... →

..., *value*

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The local variable at *index* must contain an `int`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes The *iload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

iload_<n> *iload_<n>*

Operation Load `int` from local variable

Format

iload_<n>

Forms

iload_0 = 26 (0x1a)

iload_1 = 27 (0x1b)

iload_2 = 28 (0x1c)

iload_3 = 29 (0x1d)

Operand Stack

... →

..., *value*

Description The <*n*> must be an index into the local variable array of the current frame (§2.6). The local variable at <*n*> must contain an `int`. The *value* of the local variable at <*n*> is pushed onto the operand stack.

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

imul *imul*

Operation Multiply `int`

Format

imul

Forms *imul* = 104 (0x68)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

Despite the fact that overflow may occur, execution of an *imul* instruction never throws a run-time exception.

ineg *ineg*

Operation Negate `int`

Format

ineg

Forms *ineg* = 116 (0x74)

Operand Stack ..., *value* →

..., *result*

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

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

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

instanceof *instanceof*

Operation Determine if object is of given type

Format

instanceof
indexbyte1
indexbyte2

Forms *instanceof* = 193 (0xc1)

Operand Stack ..., *objectref* →

..., *result*

Description The *objectref*, which must be of type `reference`, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a class, array, or interface type.

If *objectref* is `null`, the *instanceof* instruction pushes an `int` *result* of 0 as an `int` onto the operand stack.

Otherwise, the named class, array, or interface type is resolved (§5.4.3.1). If *objectref* is an instance of the resolved class or array type, or implements the resolved interface, the *instanceof* instruction pushes an `int` *result* of 1 as an `int` onto the operand stack; otherwise, it pushes an `int` *result* of 0.

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

- If `S` is a class type, then:
  - If `T` is a class type, then `S` must be the same class as `T`, or `S` must be a subclass of `T`;
  - If `T` is an interface type, then `S` must implement interface `T`.
- If `S` is an array type `SC[]`, that is, an array of components of type `SC`, then:
  - If `T` is a class type, then `T` must be `Object`.
  - If `T` is an interface type, then `T` must be one of the interfaces implemented by arrays (JLS §4.10.3).
  - If `T` is an array type `TC[]`, that is, an array of components of type `TC`, then one of the following must be true:
    - › `TC` and `SC` are the same primitive type.
    - › `TC` and `SC` are reference types, and type `SC` can be cast to `TC` by these run-time rules.

Linking Exceptions

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

Notes

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

invokedynamic *invokedynamic*

Operation Invoke a dynamically-computed call site

Format

invokedynamic
indexbyte1
indexbyte2
0
0

Forms *invokedynamic* = 186 (0xba)

Operand Stack ..., [*arg1*, [*arg2* ...]] →

...

Description First, the unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a dynamically-computed call site (§5.1). The values of the third and fourth operand bytes must always be zero.

The symbolic reference is resolved (§5.4.3.6) *for this specific invokedynamic instruction* to obtain a `reference` to an instance of `java.lang.invoke.CallSite`. The instance of `java.lang.invoke.CallSite` is considered "bound" to this specific *invokedynamic* instruction.

The instance of `java.lang.invoke.CallSite` indicates a *target method handle*. The *nargs* argument values are popped from the operand stack, and the target method handle is invoked. The invocation occurs as if by execution of an *invokevirtual* instruction that indicates a run-time constant pool index to a symbolic reference *R* where:

- *R* is a symbolic reference to a method of a class;
- for the symbolic reference to the class in which the method is to be found, *R* specifies `java.lang.invoke.MethodHandle`;
- for the name of the method, *R* specifies `invokeExact`;
- for the descriptor of the method, *R* specifies the method descriptor in the dynamically-computed call site.

and where it is as if the following items were pushed, in order, onto the operand stack:

- a `reference` to the target method handle;
- the *nargs* argument values, where the number, type, and order of the values must be consistent with the method descriptor in the dynamically-computed call site.

Linking Exceptions

During resolution of the symbolic reference to a dynamically-computed call site, any of the exceptions pertaining to dynamically-computed call site resolution can be thrown.

Notes

If the symbolic reference to the dynamically-computed call site can be resolved, it implies that a non-`null` `reference` to an instance of `java.lang.invoke.CallSite` is bound to the *invokedynamic* instruction. Therefore, the target method handle, indicated by the instance of `java.lang.invoke.CallSite`, is non-`null`.

Similarly, successful resolution implies that the method descriptor in the symbolic reference is semantically equal to the type descriptor of the target method handle.

Together, these invariants mean that an *invokedynamic* instruction which is bound to an instance of `java.lang.invoke.CallSite` never throws a `NullPointerException` or a `java.lang.invoke.WrongMethodTypeException`.

invokeinterface   *invokeinterface*

Operation Invoke interface method

Format

invokeinterface
indexbyte1
indexbyte2
count
0

Forms *invokeinterface* = 185 (0xb9)

Operand Stack ..., *objectref*, [*arg1*, [*arg2* ...]] →

...

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to an interface method (§5.1), which gives the name and descriptor (§4.3.3) of the interface method as well as a symbolic reference to the interface in which the interface method is to be found. The named interface method is resolved (§5.4.3.4).

The resolved interface method must not be an instance initialization method, or the class or interface initialization method (§2.9.1, §2.9.2).

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

Let `C` be the class of *objectref*. A method is selected with respect to `C` and the resolved method (§5.4.6). This is the *method to be invoked*.

If the method to be invoked is `synchronized`, the monitor associated with *objectref* is entered or reentered as if by execution of a *monitorenter* instruction (§*monitorenter*) in the current thread.

If the method to be invoked is not `native`, the *nargs* argument values and *objectref* are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The *objectref* and the argument values are consecutively made the values of local variables of the new frame, with *objectref* in local variable 0, *arg1* in local variable 1 (or, if *arg1* is of type `long` or `double`, in local variables 1 and 2), and so on. The new frame is then made current, and the Java Virtual Machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method to be invoked is `native` and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java Virtual Machine, then that is done. The *nargs* argument values and *objectref* are popped from the operand stack and are passed as parameters to the code that implements the method. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns:

- If the `native` method is `synchronized`, the monitor associated with *objectref* is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread.
- If the `native` method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the `native` method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the interface method, any of the exceptions pertaining to interface method resolution (§5.4.3.4) can be thrown.

Otherwise, if the resolved method is `static`, the *invokeinterface* instruction throws an `IncompatibleClassChangeError`.

> Note that *invokeinterface* may refer to `private` methods declared in interfaces, including nestmate interfaces.

Run-time Exceptions

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

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

Otherwise, if the selected method is neither `public` nor `private`, *invokeinterface* throws an `IllegalAccessError`.

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

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

Otherwise, if no method is selected, and there are multiple maximally-specific superinterface methods of *`C`* that match the resolved method's name and descriptor and are not `abstract`, *invokeinterface* throws an `IncompatibleClassChangeError`

Otherwise, if no method is selected, and there are no maximally-specific superinterface methods of *`C`* that match the resolved method's name and descriptor and are not `abstract`, *invokeinterface* throws an `AbstractMethodError`.

Notes

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

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

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

The selection logic allows a non-`abstract` method declared in a superinterface to be selected. Methods in interfaces are only considered if there is no matching method in the class hierarchy. In the event that there are two non-`abstract` methods in the superinterface hierarchy, with neither more specific than the other, an error occurs; there is no attempt to disambiguate (for example, one may be the referenced method and one may be unrelated, but we do not prefer the referenced method). On the other hand, if there are many `abstract` methods but only one non-`abstract` method, the non-`abstract` method is selected (unless an `abstract` method is more specific).

invokespecial *invokespecial*

Operation Invoke instance method; direct invocation of instance initialization methods and methods of the current class and its supertypes

Format

invokespecial
indexbyte1
indexbyte2

Forms *invokespecial* = 183 (0xb7)

Operand Stack ..., *objectref*, [*arg1*, [*arg2* ...]] →

...

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a method or an interface method (§5.1), which gives the name and descriptor (§4.3.3) of the method or interface method as well as a symbolic reference to the class or interface in which the method or interface method is to be found. The named method is resolved (§5.4.3.3, §5.4.3.4).

If all of the following are true, let `C` be the direct superclass of the current class:

- The resolved method is not an instance initialization method (§2.9.1).
- The symbolic reference names a class (not an interface), and that class is a superclass of the current class.
- The `ACC_SUPER` flag is set for the `class` file (§4.1).

Otherwise, let `C` be the class or interface named by the symbolic reference.

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, then it is the method to be invoked.
2. Otherwise, if `C` is a class and has a superclass, a search for a declaration of an instance method with the same name and descriptor as the resolved method is performed, starting with the direct superclass of `C` and continuing with the direct superclass of that class, and so forth, until a match is found or no further superclasses exist. If a match is found, then it is the method to be invoked.
3. Otherwise, if `C` is an interface and the class `Object` contains a declaration of a `public` instance method with the same name and descriptor as the resolved method, then it is the method to be invoked.
4. Otherwise, if there is exactly one maximally-specific method (§5.4.3.3) in the superinterfaces of `C` that matches the resolved method's name and descriptor and is not `abstract`, then it is the method to be invoked.

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

If the method is `synchronized`, the monitor associated with *objectref* is entered or reentered as if by execution of a *monitorenter* instruction (§*monitorenter*) in the current thread.

If the method is not `native`, the *nargs* argument values and *objectref* are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The *objectref* and the argument values are consecutively made the values of local variables of the new frame, with *objectref* in local variable 0, *arg1* in local variable 1 (or, if *arg1* is of type `long` or `double`, in local variables 1 and 2), and so on. The new frame is then made current, and the Java Virtual Machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is `native` and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java Virtual Machine, that is done. The *nargs* argument values and *objectref* are popped from the operand stack and are passed as parameters to the code that implements the method. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the `native` method is `synchronized`, the monitor associated with *objectref* is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread.
- If the `native` method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the `native` method and pushed onto the operand stack.

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution (§5.4.3.3) can be thrown.

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

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

Run-time Exceptions

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

Otherwise, if step 1, step 2, or step 3 of the lookup procedure selects an `abstract` method, *invokespecial* throws an `AbstractMethodError`.

Otherwise, if step 1, step 2, or step 3 of the lookup procedure selects a `native` method and the code that implements the method cannot be bound, *invokespecial* throws an `UnsatisfiedLinkError`.

Otherwise, if step 4 of the lookup procedure determines there are multiple maximally-specific superinterface methods of *`C`* that match the resolved method's name and descriptor and are not `abstract`, *invokespecial* throws an `IncompatibleClassChangeError`

Otherwise, if step 4 of the lookup procedure determines there are no maximally-specific superinterface methods of *`C`* that match the resolved method's name and descriptor and are not `abstract`, *invokespecial* throws an `AbstractMethodError`.

Notes

The difference between the *invokespecial* instruction and the *invokevirtual* instruction (§*invokevirtual*) is that *invokevirtual* invokes a method based on the class of the object. The *invokespecial* instruction is used to directly invoke instance initialization methods (§2.9.1) as well as methods of the current class and its supertypes.

> The *invokespecial* instruction was named `invokenonvirtual` prior to JDK release 1.0.2.

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

The *invokespecial* instruction handles invocation of a non-`abstract` interface method, referenced either via a direct superinterface or via a superclass. In these cases, the rules for selection are essentially the same as those for *invokeinterface* (except that the search starts from a different class).

invokestatic *invokestatic*

Operation Invoke a class (`static`) method

Format

invokestatic
indexbyte1
indexbyte2

Forms *invokestatic* = 184 (0xb8)

Operand Stack ..., [*arg1*, [*arg2* ...]] →

...

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a method or an interface method (§5.1), which gives the name and descriptor (§4.3.3) of the method or interface method as well as a symbolic reference to the class or interface in which the method or interface method is to be found. The named method is resolved (§5.4.3.3, §5.4.3.4).

The resolved method must not be an instance initialization method, or the class or interface initialization method (§2.9.1, §2.9.2).

The resolved method must be `static`, and therefore cannot be `abstract`.

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

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

If the method is `synchronized`, the monitor associated with the resolved `Class` object is entered or reentered as if by execution of a *monitorenter* instruction (§*monitorenter*) in the current thread.

If the method is not `native`, the *nargs* argument values are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The *nargs* argument values are consecutively made the values of local variables of the new frame, with *arg1* in local variable 0 (or, if *arg1* is of type `long` or `double`, in local variables 0 and 1) and so on. The new frame is then made current, and the Java Virtual Machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is `native` and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java Virtual Machine, that is done. The *nargs* argument values are popped from the operand stack and are passed as parameters to the code that implements the method. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the `native` method is `synchronized`, the monitor associated with the resolved `Class` object is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread.

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

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution (§5.4.3.3) can be thrown.

Otherwise, if the resolved method is an instance method, the *invokestatic* instruction throws an `IncompatibleClassChangeError`.

Run-time Exceptions

Otherwise, if execution of this *invokestatic* instruction causes initialization of the referenced class or interface, *invokestatic* may throw an `Error` as detailed in §5.5.

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

Notes

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

invokevirtual *invokevirtual*

Operation Invoke instance method; dispatch based on class

Format

invokevirtual
indexbyte1
indexbyte2

Forms *invokevirtual* = 182 (0xb6)

Operand Stack ..., *objectref*, [*arg1*, [*arg2* ...]] →

...

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3).

If the resolved method is not signature polymorphic (§2.9.3), then the *invokevirtual* instruction proceeds as follows.

Let `C` be the class of *objectref*. A method is selected with respect to `C` and the resolved method (§5.4.6). This is the *method to be invoked*.

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

If the method to be invoked is `synchronized`, the monitor associated with *objectref* is entered or reentered as if by execution of a *monitorenter* instruction (§*monitorenter*) in the current thread.

If the method to be invoked is not `native`, the *nargs* argument values and *objectref* are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The *objectref* and the argument values are consecutively made the values of local variables of the new frame, with *objectref* in local variable 0, *arg1* in local variable 1 (or, if *arg1* is of type `long` or `double`, in local variables 1 and 2), and so on. The new frame is then made current, and the Java Virtual Machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method to be invoked is `native` and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java Virtual Machine, that is done. The *nargs* argument values and *objectref* are popped from the operand stack and are passed as parameters to the code that implements the method. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the `native` method is `synchronized`, the monitor associated with *objectref* is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread.
- If the `native` method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the `native` method and pushed onto the operand stack.

If the resolved method is signature polymorphic (§2.9.3), *and declared in the* `java.lang.invoke.MethodHandle` *class*, then the *invokevirtual* instruction proceeds as follows, where *`D`* is the descriptor of the method symbolically referenced by the instruction.

First, a `reference` to an instance of `java.lang.invoke.MethodType` is obtained as if by resolution of a symbolic reference to a method type (§5.4.3.5) with the same parameter and return types as *`D`*.

- If the named method is `invokeExact`, the instance of `java.lang.invoke.MethodType` must be semantically equal to the type descriptor of the receiving method handle *objectref*. The *method handle to be invoked* is *objectref*.

- If the named method is `invoke`, and the instance of `java.lang.invoke.MethodType` is semantically equal to the type descriptor of the receiving method handle *objectref*, then the *method handle to be invoked* is *objectref*.
- If the named method is `invoke`, and the instance of `java.lang.invoke.MethodType` is not semantically equal to the type descriptor of the receiving method handle *objectref*, then the Java Virtual Machine attempts to adjust the type descriptor of the receiving method handle, as if by invocation of the `asType` method of `java.lang.invoke.MethodHandle`, to obtain an exactly invokable method handle *`m`*. The *method handle to be invoked* is *`m`*.

The *objectref* must be followed on the operand stack by *nargs* argument values, where the number, type, and order of the values must be consistent with the type descriptor of the method handle to be invoked. (This type descriptor will correspond to the method descriptor appropriate for the kind of the method handle to be invoked, as specified in §5.4.3.5.)

Then, if the method handle to be invoked has bytecode behavior, the Java Virtual Machine invokes the method handle as if by execution of the bytecode behavior associated with the method handle's kind. If the kind is 5 (`REF_invokeVirtual`), 6 (`REF_invokeStatic`), 7 (`REF_invokeSpecial`), 8 (`REF_newInvokeSpecial`), or 9 (`REF_invokeInterface`), then a frame will be created and made current *in the course of executing the bytecode behavior*; however, this frame is not visible, and when the method invoked by the bytecode behavior completes (normally or abruptly), the *frame of its invoker* is considered to be the frame for the method containing this *invokevirtual* instruction.

Otherwise, if the method handle to be invoked has no bytecode behavior, the Java Virtual Machine invokes it in an implementation-dependent manner.

If the resolved method is signature polymorphic and declared in the `java.lang.invoke.VarHandle` *class*, then the *invokevirtual* instruction proceeds as follows, where *`N`* and *`D`* are the name and descriptor of the method symbolically referenced by the instruction.

First, a `reference` to an instance of `java.lang.invoke.VarHandle.AccessMode` is obtained as if by invocation of the `valueFromMethodName` method of `java.lang.invoke.VarHandle.AccessMode` with a `String` argument denoting *`N`*.

Second, a `reference` to an instance of `java.lang.invoke.MethodType` is obtained as if by invocation of the `accessModeType` method of `java.lang.invoke.VarHandle` on the instance *objectref*, with the instance of `java.lang.invoke.VarHandle.AccessMode` as the argument.

Third, a `reference` to an instance of `java.lang.invoke.MethodHandle` is obtained as if by invocation of the `varHandleExactInvoker` method of `java.lang.invoke.MethodHandles` with the instance of `java.lang.invoke.VarHandle.AccessMode` as the first argument and the instance of `java.lang.invoke.MethodType` as the second argument. The resulting instance is called the *invoker method handle*.

Finally, the *nargs* argument values and *objectref* are popped from the operand stack, and the invoker method handle is invoked. The invocation occurs as if by execution of an *invokevirtual* instruction that indicates a run-time constant pool index to a symbolic reference *`R`* where:

- *`R`* is a symbolic reference to a method of a class;
- for the symbolic reference to the class in which the method is to be found, *`R`* specifies `java.lang.invoke.MethodHandle`;
- for the name of the method, *`R`* specifies `invoke`;
- for the descriptor of the method, *`R`* specifies a return type indicated by the return descriptor of *`D`*, and specifies a first parameter type of `java.lang.invoke.VarHandle` followed by

the parameter types indicated by the parameter descriptors of *D* (if any) in order.

and where it is as if the following items were pushed, in order, onto the operand stack:

- a `reference` to the instance of `java.lang.invoke.MethodHandle` (the invoker method handle);
- *objectref*;
- the *nargs* argument values, where the number, type, and order of the values must be consistent with the type descriptor of the invoker method handle.

Linking Exceptions

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution (§5.4.3.3) can be thrown.

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

Otherwise, if the resolved method is signature polymorphic and declared in the `java.lang.invoke.MethodHandle` class, then during resolution of the method type derived from the descriptor in the symbolic reference to the method, any of the exceptions pertaining to method type resolution (§5.4.3.5) can be thrown.

Otherwise, if the resolved method is signature polymorphic and declared in the `java.lang.invoke.VarHandle` class, then any linking exception that may arise from invocation of the invoker method handle can be thrown. No linking exceptions are thrown from invocation of the `valueFromMethodName`, `accessModeType`, and `varHandleExactInvoker` methods.

Run-time Exceptions

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

Otherwise, if the resolved method is not signature polymorphic:

- If the selected method is `abstract`, *invokevirtual* throws an `AbstractMethodError`.
- Otherwise, if the selected method is `native` and the code that implements the method cannot be bound, *invokevirtual* throws an `UnsatisfiedLinkError`.
- Otherwise, if no method is selected, and there are multiple maximally-specific superinterface methods of *C* that match the resolved method's name and descriptor and are not `abstract`, *invokevirtual* throws an `IncompatibleClassChangeError`
- Otherwise, if no method is selected, and there are no maximally-specific superinterface methods of *C* that match the resolved method's name and descriptor and are not `abstract`, *invokevirtual* throws an `AbstractMethodError`.

Otherwise, if the resolved method is signature polymorphic and declared in the `java.lang.invoke.MethodHandle` class, then:

- If the method name is `invokeExact`, and the obtained instance of `java.lang.invoke.MethodType` is not semantically equal to the type descriptor of the receiving method handle *objectref*, the *invokevirtual* instruction throws a `java.lang.invoke.WrongMethodTypeException`.
- If the method name is `invoke`, and the obtained instance of `java.lang.invoke.MethodType` is not a valid argument to the `asType` method of `java.lang.invoke.MethodHandle` invoked on the receiving method handle *objectref*, the *invokevirtual* instruction throws a `java.lang.invoke.WrongMethodTypeException`.

Otherwise, if the resolved method is signature polymorphic and declared in the `java.lang.invoke.VarHandle` class, then any run-time exception that may arise from invocation of the invoker method handle can be thrown. No run-time exceptions are thrown from invocation of the `valueFromMethodName`, `accessModeType`, and `varHandleExactInvoker` methods, except `NullPointerException` if *objectref* is `null`.

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

It is possible that the symbolic reference of an *invokevirtual* instruction resolves to an interface method. In this case, it is possible that there is no overriding method in the class hierarchy, but that a non-`abstract` interface method matches the resolved method's descriptor. The selection logic matches such a method, using the same rules as for *invokeinterface*.

ior *ior*

Operation Boolean OR `int`

Format

ior

Forms *ior* = 128 (0x80)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

irem *irem*

Operation Remainder `int`

Format

irem

Forms *irem* = 112 (0x70)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

Run-time Exception If the value of the divisor for an `int` remainder operator is 0, *irem* throws an `ArithmeticException`.

ireturn *ireturn*

Operation Return `int` from method

Format | *ireturn* |

Forms *ireturn* = 172 (0xac)

Operand Stack ..., *value* →

[empty]

Description The current method must have return type `boolean`, `byte`, `char`, `short`, or `int`. The *value* must be of type `int`. If the current method is a `synchronized` method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread. If no exception is thrown, *value* is popped from the operand stack of the current frame (§2.6) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

Prior to pushing *value* onto the operand stack of the frame of the invoker, it may have to be converted. If the return type of the invoked method was `byte`, `char`, or `short`, then *value* is converted from `int` to the return type as if by execution of *i2b*, *i2c*, or *i2s*, respectively. If the return type of the invoked method was `boolean`, then *value* is narrowed from `int` to `boolean` by taking the bitwise AND of *value* and 1.

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

Run-time Exceptions If the Java Virtual Machine implementation does not enforce the rules on structured locking described in §2.11.10, then if the current method is a `synchronized` method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, *ireturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a `synchronized` method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is synchronized.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the first of those rules is violated during invocation of the current method, then *ireturn* throws an `IllegalMonitorStateException`.

ishl *ishl*

Operation Shift left `int`

Format *ishl*

Forms *ishl* = 120 (0x78)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

ishr *ishr*

Operation Arithmetic shift right `int`

Format

ishr

Forms *ishr* = 122 (0x7a)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

Notes The resulting value is *floor*(*value1* / 2^s), where *s* is *value2* & 0x1f. For non-negative *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.

istore *istore*

Operation Store `int` into local variable

Format

istore
index

Forms *istore* = 54 (0x36)

Operand Stack ..., *value* →

...

Description The *index* is an unsigned byte that must be an index into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, and the value of the local variable at *index* is set to *value*.

Notes The *istore* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

istore_<n> ***istore_<n>***

Operation Store `int` into local variable

Format

istore_<n>

Forms

istore_0 = 59 (0x3b)

istore_1 = 60 (0x3c)

istore_2 = 61 (0x3d)

istore_3 = 62 (0x3e)

Operand Stack

..., *value* →

...

Description The <*n*> must be an index into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *value*.

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

isub *isub*

Operation Subtract `int`

Format

isub

Forms *isub* = 100 (0x64)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

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

Despite the fact that overflow may occur, execution of an *isub* instruction never throws a run-time exception.

iushr ***iushr***

Operation Logical shift right `int`

Format

iushr

Forms *iushr* = 124 (0x7c)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

Notes If *value1* is positive and *s* 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.

ixor *ixor*

Operation Boolean XOR `int`

Format

ixor

Forms *ixor* = 130 (0x82)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

jsr

Operation Jump subroutine

Format

jsr
branchbyte1
branchbyte2

Forms *jsr* = 168 (0xa8)

Operand Stack ... →

..., *address*

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

Notes Note that *jsr* pushes the address onto the operand stack and *ret* (§*ret*) gets it out of a local variable. This asymmetry is intentional.

In Oracle's implementation of a compiler for the Java programming language prior to Java SE 6, the *jsr* instruction was used with the *ret* instruction in the implementation of the `finally` clause (§3.13, §4.10.2.5).

jsr_w *jsr_w*

Operation Jump subroutine (wide index)

Format

jsr_w
branchbyte1
branchbyte2
branchbyte3
branchbyte4

Forms *jsr_w* = 201 (0xc9)

Operand Stack ... →

..., *address*

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

Notes Note that *jsr_w* pushes the address onto the operand stack and *ret* (§*ret*) gets it out of a local variable. This asymmetry is intentional.

In Oracle's implementation of a compiler for the Java programming language prior to Java SE 6, the *jsr_w* instruction was used with the *ret* instruction in the implementation of the `finally` clause (§3.13, §4.10.2.5).

Although the *jsr_w* instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes (§4.11). This limit may be raised in a future release of the Java Virtual Machine.

l2d *l2d*

Operation Convert `long` to `double`

Format

l2d

Forms *l2d* = 138 (0x8a)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to a `double` *result* using the round to nearest rounding policy (§2.8). The *result* is pushed onto the operand stack.

Notes The *l2d* instruction performs a widening primitive conversion (JLS §5.1.2) that may lose precision because values of type `double` have only 53 significand bits.

l2f *l2f*

Operation Convert `long` to `float`

Format

l2f

Forms *l2f* = 137 (0x89)

Operand Stack ..., *value* →

..., *result*

Description The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to a `float` *result* using the round to nearest rounding policy (§2.8). The *result* is pushed onto the operand stack.

Notes The *l2f* instruction performs a widening primitive conversion (JLS §5.1.2) that may lose precision because values of type `float` have only 24 significand bits.

l2i ***l2i***

Operation Convert `long` to `int`

Format

l2i

Forms *l2i* = 136 (0x88)

Operand Stack

..., *value* →

..., *result*

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

Notes The *l2i* instruction performs a narrowing primitive conversion (JLS §5.1.3). It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as value.

ladd *ladd*

Operation Add `long`

Format

ladd

Forms *ladd* = 97 (0x61)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

Despite the fact that overflow may occur, execution of an *ladd* instruction never throws a run-time exception.

laload *laload*

Operation Load `long` from array

Format

laload

Forms *laload* = 47 (0x2f)

Operand Stack ..., *arrayref*, *index* →

..., *value*

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

Run-time Exceptions If *arrayref* is `null`, *laload* throws a `NullPointerException`.

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

land *land*

Operation Boolean AND `long`

Format *land*

Forms *land* = 127 (0x7f)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

lastore

Operation Store into `long` array

Format

lastore

Forms *lastore* = 80 (0x50)

Operand Stack ..., *arrayref*, *index*, *value* →

...

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

Run-time Exceptions If *arrayref* is `null`, *lastore* throws a `NullPointerException`.

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

lcmp *lcmp*

Operation Compare `long`

Format

lcmp

Forms *lcmp* = 148 (0x94)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

lconst_<l> *lconst_<l>*

Operation Push `long` constant

Format

lconst_<l>

Forms *lconst_0* = 9 (0x9)

lconst_1 = 10 (0xa)

Operand Stack ... →

..., *<l>*

Description Push the `long` constant *<l>* (0 or 1) onto the operand stack.

ldc *ldc*

Operation Push item from run-time constant pool

Format

ldc
index

Forms *ldc* = 18 (0x12)

Operand Stack ... →

..., *value*

Description The *index* is an unsigned byte that must be a valid index into the run-time constant pool of the current class (§2.5.5). The run-time constant pool entry at *index* must be loadable (§5.1), and not any of the following:

- A numeric constant of type `long` or `double`.
- A symbolic reference to a dynamically-computed constant whose field descriptor is `J` (denoting `long`) or `D` (denoting `double`).

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

Otherwise, if the run-time constant pool entry is a string constant, that is, a `reference` to an instance of class `String`, then *value*, a `reference` to that instance, is pushed onto the operand stack.

Otherwise, if the run-time constant pool entry is a symbolic reference to a class or interface, then the named class or interface is resolved (§5.4.3.1) and *value*, a `reference` to the `Class` object representing that class or interface, is pushed onto the operand stack.

Otherwise, the run-time constant pool entry is a symbolic reference to a method type, a method handle, or a dynamically-computed constant. The symbolic reference is resolved (§5.4.3.5,

§5.4.3.6) and *value*, the result of resolution, is pushed onto the operand stack.

Linking Exceptions

During resolution of a symbolic reference, any of the exceptions pertaining to resolution of that kind of symbolic reference can be thrown.

ldc_w *ldc_w*

Operation Push item from run-time constant pool (wide index)

Format

ldc_w
indexbyte1
indexbyte2

Forms *ldc_w* = 19 (0x13)

Operand Stack ... →

..., *value*

Description The unsigned *indexbyte1* and *indexbyte2* are assembled into an unsigned 16-bit index into the run-time constant pool of the current class (§2.5.5), where the value of the index is calculated as (*indexbyte1* << 8) | *indexbyte2*. The index must be a valid index into the run-time constant pool of the current class. The run-time constant pool entry at the index must be loadable (§5.1), and not any of the following:

- A numeric constant of type `long` or `double`.
- A symbolic reference to a dynamically-computed constant whose field descriptor is `J` (denoting `long`) or `D` (denoting `double`).

If the run-time constant pool entry is a numeric constant of type `int` or `float`, or a string constant, then *value* is determined and pushed onto the operand stack according to the rules given for the *ldc* instruction.

Otherwise, the run-time constant pool entry is a symbolic reference to a class, interface, method type, method handle, or dynamically-computed constant. It is resolved and *value* is determined and pushed onto the operand stack according to the rules given for the *ldc* instruction.

Linking Exceptions

During resolution of a symbolic reference, any of the exceptions pertaining to resolution of that kind of symbolic reference can be thrown.

Notes

The *ldc_w* instruction is identical to the *ldc* instruction (§*ldc*) except for its wider run-time constant pool index.

ldc2_w

Operation

Push `long` or `double` from run-time constant pool (wide index)

Format

ldc2_w
indexbyte1
indexbyte2

Forms

ldc2_w = 20 (0x14)

Operand Stack

... →

..., *value*

Description

The unsigned *indexbyte1* and *indexbyte2* are assembled into an unsigned 16-bit index into the run-time constant pool of the current class (§2.5.5), where the value of the index is calculated as (*indexbyte1* << 8) | *indexbyte2*. The index must be a valid index into the run-time constant pool of the current class. The run-time constant pool entry at the index must be loadable (§5.1), and in particular one of the following:

- A numeric constant of type `long` or `double`.
- A symbolic reference to a dynamically-computed constant whose field descriptor is `J` (denoting `long`) or `D` (denoting `double`).

If the run-time constant pool entry is a numeric constant of type `long` or `double`, then the *value* of that numeric constant is pushed onto the operand stack as a `long` or `double`, respectively.

Otherwise, the run-time constant pool entry is a symbolic reference to a dynamically-computed constant. The symbolic reference is resolved (§5.4.3.6) and *value*, the result of resolution, is pushed onto the operand stack.

Linking Exceptions

During resolution of a symbolic reference to a dynamically-computed constant, any of the exceptions pertaining to dynamically-computed constant resolution can be thrown.

Notes Only a wide-index version of the *ldc2_w* instruction exists; there is no *ldc2* instruction that pushes a `long` or `double` with a single-byte index.

ldiv *ldiv*

Operation Divide `long`

Format

ldiv

Forms *ldiv* = 109 (0x6d)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

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

Run-time Exception If the value of the divisor in a `long` division is 0, *ldiv* throws an `ArithmeticException`.

lload ***lload***

Operation Load `long` from local variable

Format

lload
index

Forms *lload* = 22 (0x16)

Operand Stack ... →

..., *value*

Description The *index* is an unsigned byte. Both *index* and *index*+1 must be indices into the local variable array of the current frame (§2.6). The local variable at *index* must contain a `long`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes The *lload* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

lload_<n> *lload_<n>*

Operation Load `long` from local variable

Format

lload_<n>

Forms *lload_0* = 30 (0x1e)

lload_1 = 31 (0x1f)

lload_2 = 32 (0x20)

lload_3 = 33 (0x21)

Operand Stack ... →

..., *value*

Description Both <*n*> and <*n*>+1 must be indices into the local variable array of the current frame (§2.6). The local variable at <*n*> must contain a `long`. The *value* of the local variable at <*n*> is pushed onto the operand stack.

Notes Each of the *lload_<n>* instructions is the same as *lload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

lmul

Operation Multiply `long`

Format

lmul

Forms *lmul* = 105 (0x69)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

Despite the fact that overflow may occur, execution of an *lmul* instruction never throws a run-time exception.

lneg *lneg*

Operation Negate `long`

Format

lneg

Forms *lneg* = 117 (0x75)

Operand Stack ..., *value* →

..., *result*

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

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

For all `long` values `x`, `-x` equals `(~x)+1`.

lookupswitch ***lookupswitch***

Operation Access jump table by key match and jump

Format

lookupswitch
<0-3 byte pad>
defaultbyte1
defaultbyte2
defaultbyte3
defaultbyte4
npairs1
npairs2
npairs3
npairs4
match-offset pairs...

Forms *lookupswitch* = 171 (0xab)

Operand Stack ..., *key* →

...

Description A *lookupswitch* is a variable-length instruction. Immediately after the *lookupswitch* opcode, between zero and three bytes must act as padding, such 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 follow a series of signed 32-bit values: *default*, *npairs*, and then *npairs* pairs of signed 32-bit values. The *npairs* must be greater than or equal to 0. Each of the *npairs* pairs consists of an `int` *match* and a signed 32-bit *offset*. Each of these signed 32-bit values is constructed from four unsigned bytes as (*byte1* << 24) | (*byte2* << 16) | (*byte3* << 8) | *byte4*.

The table *match-offset* pairs of the *lookupswitch* instruction must be sorted in increasing numerical order by *match*.

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

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

Notes

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

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

lor ***lor***

Operation Boolean OR `long`

Format

lor

Forms *lor* = 129 (0x81)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

lrem *lrem*

Operation Remainder `long`

Format

lrem

Forms *lrem* = 113 (0x71)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

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

lreturn *lreturn*

Operation Return `long` from method

Format

lreturn

Forms *lreturn* = 173 (0xad)

Operand Stack ..., *value* →

[empty]

Description The current method must have return type `long`. The *value* must be of type `long`. If the current method is a `synchronized` method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread. If no exception is thrown, *value* is popped from the operand stack of the current frame (§2.6) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

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

Run-time Exceptions If the Java Virtual Machine implementation does not enforce the rules on structured locking described in §2.11.10, then if the current method is a `synchronized` method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, *lreturn* throws an `IllegalMonitorStateException`. This can happen, for example, if a `synchronized` method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is `synchronized`.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the first of those rules is violated during invocation of the current method, then *lreturn* throws an `IllegalMonitorStateException`.

lshl *lshl*

Operation Shift left `long`

Format

lshl

Forms *lshl* = 121 (0x79)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

lshr ***lshr***

Operation Arithmetic shift right `long`

Format

lshr

Forms *lshr* = 123 (0x7b)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

Notes The resulting value is *floor*(*value1* / 2^s), where *s* is *value2* & 0x3f. For non-negative *value1*, this is equivalent to truncating `long` division by 2 to the power *s*. The shift distance actually used is therefore always in the range 0 to 63, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x3f.

lstore *lstore*

Operation Store `long` into local variable

Format

lstore
index

Forms *lstore* = 55 (0x37)

Operand Stack ..., *value* →

...

Description The *index* is an unsigned byte. Both *index* and *index*+1 must be indices into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at *index* and *index*+1 are set to *value*.

Notes The *lstore* opcode can be used in conjunction with the *wide* instruction (§*wide*) to access a local variable using a two-byte unsigned index.

lstore_<n> ***lstore_<n>***

Operation Store `long` into local variable

Format

lstore_<n>

Forms *lstore_0* = 63 (0x3f)

lstore_1 = 64 (0x40)

lstore_2 = 65 (0x41)

lstore_3 = 66 (0x42)

Operand Stack ..., *value* →

...

Description Both <*n*> and <*n*>+1 must be indices into the local variable array of the current frame (§2.6). The *value* on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at <*n*> and <*n*>+1 are set to *value*.

Notes Each of the *lstore_<n>* instructions is the same as *lstore* with an *index* of <*n*>, except that the operand <*n*> is implicit.

lsub *lsub*

Operation Subtract `long`

Format

lsub

Forms *lsub* = 101 (0x65)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

For `long` subtraction, `a-b` produces the same result 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 type `long`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical difference of the two values.

Despite the fact that overflow may occur, execution of an *lsub* instruction never throws a run-time exception.

lushr ***lushr***

Operation Logical shift right `long`

Format

lushr

Forms *lushr* = 125 (0x7d)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

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

lxor *lxor*

Operation Boolean XOR `long`

Format

lxor

Forms *lxor* = 131 (0x83)

Operand Stack ..., *value1*, *value2* →

..., *result*

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

monitorenter *monitorenter*

Operation Enter monitor for object

Format

monitorenter

Forms *monitorenter* = 194 (0xc2)

Operand Stack ..., *objectref* →

...

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

Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes *monitorenter* attempts to gain ownership of the monitor associated with *objectref*, as follows:

- If the entry count of the monitor associated with *objectref* is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.
- If the thread already owns the monitor associated with *objectref*, it reenters the monitor, incrementing its entry count.
- If another thread already owns the monitor associated with *objectref*, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

Run-time Exception If *objectref* is `null`, *monitorenter* throws a `NullPointerException`.

Notes A *monitorenter* instruction may be used with one or more *monitorexit* instructions (§*monitorexit*) to implement a `synchronized` statement in the Java programming language (§3.14). The *monitorenter* and *monitorexit* instructions are not used in the implementation of `synchronized` methods, although they can be used to provide equivalent locking semantics. Monitor entry on invocation of a `synchronized` method, and monitor exit on its return, are handled implicitly by the Java Virtual Machine's method invocation and return instructions, as if *monitorenter* and *monitorexit* were used.

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

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

monitorexit *monitorexit*

Operation Exit monitor for object

Format

monitorexit

Forms *monitorexit* = 195 (0xc3)

Operand Stack ..., *objectref* →

...

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

The thread that executes *monitorexit* must be the owner of the monitor associated with the instance referenced by *objectref*.

The thread decrements the entry count of the monitor associated with *objectref*. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

Run-time Exceptions If *objectref* is `null`, *monitorexit* throws a `NullPointerException`.

Otherwise, if the thread that executes *monitorexit* is not the owner of the monitor associated with the instance referenced by *objectref*, *monitorexit* throws an `IllegalMonitorStateException`.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the second of those rules is violated by the execution of this *monitorexit* instruction, then *monitorexit* throws an `IllegalMonitorStateException`.

Notes One or more *monitorexit* instructions may be used with a *monitorenter* instruction (§*monitorenter*) to implement a `synchronized` statement in the Java programming language (§3.14). The *monitorenter* and *monitorexit* instructions are not used in the implementation of `synchronized` methods, although they can be used to provide equivalent locking semantics.

The Java Virtual Machine supports exceptions thrown within `synchronized` methods and `synchronized` statements differently:

- Monitor exit on normal `synchronized` method completion is handled by the Java Virtual Machine's return instructions. Monitor exit on abrupt `synchronized` method completion is handled implicitly by the Java Virtual Machine's *athrow* instruction.
- When an exception is thrown from within a `synchronized` statement, exit from the monitor entered prior to the execution of the `synchronized` statement is achieved using the Java Virtual Machine's exception handling mechanism (§3.14).

multianewarray

Operation Create new multidimensional array

Format

multianewarray
indexbyte1
indexbyte2
dimensions

Forms *multianewarray* = 197 (0xc5)

Operand Stack ..., *count1*, [*count2*, ...] →

..., *arrayref*

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

All of the *count* values are popped off the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1). The resulting entry must be an array class type of dimensionality greater than or equal to *dimensions*.

A new multidimensional array of the array type is allocated from the garbage-collected heap. If any *count* value is zero, no subsequent dimensions are allocated. The components of the array in the first dimension are initialized to subarrays of the type of the second dimension, and so on. The components of the last allocated dimension of the array are initialized to the default initial value

(§2.3, §2.4) for the element type of the array type. A `reference` *arrayref* to the new array is pushed onto the operand stack.

Linking Exceptions

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

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

Run-time Exception

Otherwise, if any of the *dimensions* values on the operand stack are less than zero, the *multianewarray* instruction throws a `NegativeArraySizeException`.

Notes

It may be more efficient to use *newarray* or *anewarray* (§*newarray*, §*anewarray*) when creating an array of a single dimension.

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

new *new*

Operation Create new object

Format

new
indexbyte1
indexbyte2

Forms *new* = 187 (0xbb)

Operand Stack ... →

..., *objectref*

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a class or interface type. The named class or interface type is resolved (§5.4.3.1) and should result in a class type. Memory for a new instance of that class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized to their default initial values (§2.3, §2.4). The *objectref*, a `reference` to the instance, is pushed onto the operand stack.

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

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

Otherwise, if the symbolic reference to the class or interface type resolves to an interface or an `abstract` class, *new* throws an `InstantiationError`.

Run-time Exception

Otherwise, if execution of this *new* instruction causes initialization of the referenced class, *new* may throw an `Error` as detailed in JLS §15.9.4.

Notes

The *new* instruction does not completely create a new instance; instance creation is not completed until an instance initialization method (§2.9.1) has been invoked on the uninitialized instance.

newarray *newarray*

Operation Create new array

Format

newarray
atype

Forms *newarray* = 188 (0xbc)

Operand Stack ..., *count* →

..., *arrayref*

Description The *count* must be of type `int`. It is popped off the operand stack. The *count* represents the number of elements in the array to be created.

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

Table 6.5.newarray-A. Array type codes

Array Type	***atype***
`T_BOOLEAN`	4
`T_CHAR`	5
`T_FLOAT`	6
`T_DOUBLE`	7
`T_BYTE`	8
`T_SHORT`	9
`T_INT`	10
`T_LONG`	11

A new array whose components are of type *atype* and of length *count* is allocated from the garbage-collected heap. A `reference` *arrayref* to this new array object is pushed into the operand stack. Each of the elements of the new array is initialized to the default initial value (§2.3, §2.4) for the element type of the array type.

Run-time Exception If *count* is less than zero, *newarray* throws a `NegativeArraySizeException`.

Notes In Oracle's Java Virtual Machine implementation, arrays of type `boolean` (*atype* is `T_BOOLEAN`) are stored as arrays of 8-bit values and are manipulated using the *baload* and *bastore* instructions (§*baload*, §*bastore*) which also access arrays of type `byte`. Other implementations may implement packed `boolean` arrays; the *baload* and *bastore* instructions must still be used to access those arrays.

nop *nop*

Operation Do nothing

Format

nop

Forms *nop* = 0 (0x0)

Operand Stack No change

Description Do nothing.

pop *pop*

Operation Pop the top operand stack value

Format

pop

Forms *pop* = 87 (0x57)

Operand Stack ..., *value* →

...

Description Pop the top value from the operand stack.

The *pop* instruction must not be used unless *value* is a value of a category 1 computational type (§2.11.1).

pop2 ***pop2***

Operation Pop the top one or two operand stack values

Format

pop2

Forms *pop2* = 88 (0x58)

Operand Stack

Form 1:

..., *value2*, *value1* →

...

where each of *value1* and *value2* is a value of a category 1 computational type (§2.11.1).

Form 2:

..., *value* →

...

where *value* is a value of a category 2 computational type (§2.11.1).

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

putfield *putfield*

Operation Set field in object

Format

putfield
indexbyte1
indexbyte2

Forms *putfield* = 181 (0xb5)

Operand Stack ..., *objectref*, *value* →

...

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.2).

The type of a *value* stored by a *putfield* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type. If the field is `final`, it must be declared in the current class, and the instruction must occur in an instance initialization method of the current class (§2.9.1).

The *value* and *objectref* are popped from the operand stack.

The *objectref* must be of type `reference` but not an array type.

If the *value* is of type `int` and the field descriptor type is `boolean`, then the `int` *value* is narrowed by taking the bitwise AND of *value* and 1, resulting in *value*'. The referenced field in *objectref* is set to *value*'.

Otherwise, the referenced field in *objectref* is set to *value*.

Linking Exceptions

During resolution of the symbolic reference to the field, any of the exceptions pertaining to field resolution (§5.4.3.2) can be thrown.

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

Otherwise, if the resolved field is `final`, it must be declared in the current class, and the instruction must occur in an instance initialization method of the current class. Otherwise, an `IllegalAccessError` is thrown.

Run-time Exception

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

putstatic *putstatic*

Operation Set static field in class

Format

putstatic
indexbyte1
indexbyte2

Forms *putstatic* = 179 (0xb3)

Operand Stack ..., *value* →

...

Description The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool entry at the index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved (§5.4.3.2).

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

The type of a *value* stored by a *putstatic* instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the *value* must be an `int`. If the field descriptor type is `float`, `long`, or `double`, then the *value* must be a `float`, `long`, or `double`, respectively. If the field descriptor type is a reference type, then the *value* must be of a type that is assignment compatible (JLS §5.2) with the field descriptor type. If the field is `final`, it must be declared in the current class or interface, and the instruction must occur in the class or interface initialization method of the current class or interface (§2.9.2).

The *value* is popped from the operand stack.

If the *value* is of type `int` and the field descriptor type is `boolean`, then the `int` *value* is narrowed by taking the bitwise AND of *value* and 1, resulting in *value*'. The referenced field in the class or interface is set to *value*'.

Otherwise, the referenced field in the class or interface is set to *value*.

Linking Exceptions

During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution (§5.4.3.2) can be thrown.

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

Otherwise, if the resolved field is `final`, it must be declared in the current class or interface, and the instruction must occur in the class or interface initialization method of the current class or interface. Otherwise, an `IllegalAccessError` is thrown.

Run-time Exception

Otherwise, if execution of this *putstatic* instruction causes initialization of the referenced class or interface, *putstatic* may throw an `Error` as detailed in §5.5.

Notes

A *putstatic* instruction may be used only to set the value of an interface field on the initialization of that field. Interface fields may be assigned to only once, on execution of an interface variable initialization expression when the interface is initialized (§5.5, JLS §9.3.1).

ret

Operation Return from subroutine

Format

ret
index

Forms *ret* = 169 (0xa9)

Operand Stack No change

Description The *index* is an unsigned byte between 0 and 255, inclusive. The local variable at *index* in the current frame (§2.6) must contain a value of type `returnAddress`. The contents of the local variable are written into the Java Virtual Machine's `pc` register, and execution continues there.

Notes Note that *jsr* (§*jsr*) pushes the address onto the operand stack and *ret* gets it out of a local variable. This asymmetry is intentional.

In Oracle's implementation of a compiler for the Java programming language prior to Java SE 6, the *ret* instruction was used with the *jsr* and *jsr_w* instructions (§*jsr*, §*jsr_w*) in the implementation of the `finally` clause (§3.13, §4.10.2.5).

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

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

return

Operation Return `void` from method

Format

return

Forms *return* = 177 (0xb1)

Operand Stack

... →

[empty]

Description The current method must have return type `void`. If the current method is a `synchronized` method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a *monitorexit* instruction (§*monitorexit*) in the current thread. If no exception is thrown, any values on the operand stack of the current frame (§2.6) are discarded.

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

Run-time Exceptions If the Java Virtual Machine implementation does not enforce the rules on structured locking described in §2.11.10, then if the current method is a `synchronized` method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, *return* throws an `IllegalMonitorStateException`. This can happen, for example, if a `synchronized` method contains a *monitorexit* instruction, but no *monitorenter* instruction, on the object on which the method is `synchronized`.

Otherwise, if the Java Virtual Machine implementation enforces the rules on structured locking described in §2.11.10 and if the first of those rules is violated during invocation of the current method, then *return* throws an `IllegalMonitorStateException`.

saload *saload*

Operation Load `short` from array

Format *saload*

Forms *saload* = 53 (0x35)

Operand Stack ..., *arrayref*, *index* →

..., *value*

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

Run-time Exceptions If *arrayref* is `null`, *saload* throws a `NullPointerException`.

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

sastore *sastore*

Operation Store into `short` array

Format

sastore

Forms *sastore* = 86 (0x56)

Operand Stack ..., *arrayref*, *index*, *value* →

...

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

Run-time Exceptions If *arrayref* is `null`, *sastore* throws a `NullPointerException`.

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

sipush ***sipush***

Operation Push `short`

Format

sipush
byte1
byte2

Forms *sipush* = 17 (0x11)

Operand Stack ... →

..., *value*

Description The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate `short`, where the value of the `short` is (*byte1* << 8) | *byte2*. The intermediate value is then sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.

swap *swap*

Operation Swap the top two operand stack values

Format

swap

Forms *swap* = 95 (0x5f)

Operand Stack ..., *value2*, *value1* →

..., *value1*, *value2*

Description Swap the top two values on the operand stack.

The *swap* instruction must not be used unless *value1* and *value2* are both values of a category 1 computational type (§2.11.1).

Notes The Java Virtual Machine does not provide an instruction implementing a swap on operands of category 2 computational types.

tableswitch *tableswitch*

Operation Access jump table by index and jump

Format

tableswitch
<0-3 byte pad>
defaultbyte1
defaultbyte2
defaultbyte3
defaultbyte4
lowbyte1
lowbyte2
lowbyte3
lowbyte4
highbyte1
highbyte2
highbyte3
highbyte4
jump offsets...

Forms *tableswitch* = 170 (0xaa)

Operand Stack ..., *index* →

...

Description A *tableswitch* is a variable-length instruction. Immediately after the *tableswitch* opcode, between zero and three bytes must act as padding, such 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 are bytes constituting three signed 32-bit values: *default*, *low*, and *high*. Immediately following are bytes constituting a series of *high* - *low* + 1 signed 32-bit offsets. The value *low* must be less than or equal to *high*. The *high* - *low* + 1 signed 32-bit offsets are treated as a 0-based jump table. Each of these signed 32-bit values is constructed as (*byte1* << 24) | (*byte2* << 16) | (*byte3* << 8) | *byte4*.

The *index* must be of type `int` and is popped from the operand stack. If *index* is less than *low* or *index* is greater than *high*, then a target address is calculated by adding *default* to the address of the opcode of this *tableswitch* instruction. Otherwise, the offset at position *index* - *low* of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this *tableswitch* instruction. Execution then continues at the target address.

The target address that can be calculated from each jump table offset, as well as the one that can be calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *tableswitch* instruction.

Notes

The alignment required of the 4-byte operands of the *tableswitch* instruction guarantees 4-byte alignment of those operands if and only if the method that contains the *tableswitch* starts on a 4-byte boundary.

wide *wide*

Operation Extend local variable index by additional bytes

Format 1

wide
<opcode>
indexbyte1
indexbyte2

where *<opcode>* is one of *iload*, *fload*, *aload*, *lload*, *dload*, *istore*, *fstore*, *astore*, *lstore*, *dstore*, or *ret*

Format 2

wide
iinc
indexbyte1
indexbyte2
constbyte1
constbyte2

Forms *wide* = 196 (0xc4)

Operand Stack Same as modified instruction

Description The *wide* instruction modifies the behavior of another instruction. It takes one of two formats, depending on the instruction being modified. The first form of the *wide* instruction modifies one of the instructions *iload*, *fload*, *aload*, *lload*, *dload*, *istore*, *fstore*, *astore*, *lstore*, *dstore*, or *ret* (§*iload*, §*fload*, §*aload*, §*lload*, §*dload*, §*istore*, §*fstore*, §*astore*, §*lstore*, §*dstore*, §*ret*). The second form applies only to the *iinc* instruction (§*iinc*).

In either case, the *wide* opcode itself is followed in the compiled code by the opcode of the instruction *wide* modifies. In either form, two unsigned bytes *indexbyte1* and *indexbyte2* follow the modified opcode and are assembled into a 16-bit unsigned index to a local variable in the current frame (§2.6), where the value