CHAPTER 7

# Java Card Virtual Machine Instruction Set

A Java Card virtual machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. This chapter gives details about the format of each Java Card virtual machine instruction and the operation it performs.

## 7.1 Assumptions: The Meaning of “Must”

The description of each instruction is always given in the context of Java Card virtual machine code that satisfies the static and structural constraints of Chapter 6, “The CAP File Format”.

In the description of individual Java Card virtual machine instructions, we frequently state that some situation “must” or “must not” be the case: “The value2 must be of type int.” The constraints of Chapter 6 “The CAP File Format” guarantee that all such expectations will in fact be met. If some constraint (a “must” or “must not”) in an instruction description is not satisfied at run time, the behavior of the Java Card virtual machine is undefined.

## 7.2 Reserved Opcodes

In addition to the opcodes of the instructions specified later this chapter, which are used in Java Card CAP files (see Chapter 6, “The CAP File Format”), two opcodes are reserved for internal use by a Java Card virtual machine implementation. If Sun Microsystems, Inc. extends the instruction set of the Java Card virtual machine in the future, these reserved opcodes are guaranteed not to be used.

The two reserved opcodes, numbers 254 (0xfe) and 255 (0xff), have the mnemonics impdep1 and impdep2, respectively. These instructions are intended to provide ”back doors” or traps to implementation-specific functionality implemented in software and hardware, respectively.

Although these opcodes have been reserved, they may only be used inside a Java Card virtual machine implementation. They cannot appear in valid CAP files.

## 7.3 Virtual Machine Errors

A Java Card virtual machine may encounter internal errors or resource limitations that prevent it from executing correctly written Java programs. While The Java Virtual Machine Specification allows reporting and handling of virtual machine errors, it also states that they cannot ordinarily be handled by application code. This Virtual Machine Specification for the Java Card Platform, v2.2.2 is more restrictive in that it does not allow for any reporting or handling of unrecoverable virtual machine errors at the application code level. A virtual machine error is considered unrecoverable if further execution could compromise the security or correct operation of the virtual machine or underlying system software. When an unrecoverable error occurs, the virtual machine will halt bytecode execution. Responses beyond halting the virtual machine are implementation-specific policies and are not mandated in this specification.

In the case where the virtual machine encounters a recoverable error, such as insufficient memory to allocate a new object, it will throw a SystemException with an error code describing the error condition. The Virtual Machine Specification for the Java Card Platform, v2.2.2 cannot predict where resource limitations or internal errors may be encountered and does not mandate precisely when they can be reported. Thus, a SystemException may be thrown at any time during the operation of the Java Card virtual machine.

## 7.4 Security Exceptions

Instructions of the Java Card virtual machine throw an instance of the class SecurityException when a security violation has been detected. The Java Card virtual machine does not mandate the complete set of security violations that can or will result in an exception being thrown. However, there is a minimum set that must be supported.

In the general case, any instruction that de-references an object reference must throw a SecurityException if the context (Section 3.4, “Contexts” on page 3-2) in which the instruction is executing is different than the owning context (Section 3.4, “Contexts” on page 3-2) of the referenced object. The list of instructions includes the instance field get and put instructions, the array load and store instructions, as well as the arraylength, invokeinterface, invokespecial, invokevirtual, checkcast, instanceof and athrow instructions.

There are several exceptions to this general rule that allow cross-context use of objects or arrays. These exceptions are detailed in Chapter 6 of the *Runtime Environment Specification for the Java Card Platform, Version 2.2.2*. An important detail to note is that any cross-context method invocation will result in a context switch (Section 3.4, “Contexts” on page 3-2).

The Java Card virtual machine may also throw a SecurityException if an instruction violates any of the static constraints of Chapter 6, “The CAP File Format”. The *Virtual Machine Specification for the Java Card Platform, Version 2.2.2* does not mandate which instructions must implement these additional security checks, or to what level. Therefore, a `SecurityException` may be thrown at any time during the operation of the Java Card virtual machine.

## 7.5 The Java Card Virtual Machine Instruction Set

Java virtual machine instructions are represented in this chapter by entries of the form shown in TABLE 7-1, an example instruction page, in alphabetical order.

**TABLE 7-1** Example Instruction Page

*mnemonic*

Short description of the instruction.

**Format**

*mnemonic*
*operand1*
*operand2*
...

**Forms**

*mnemonic = opcode*

**Stack**

*..., value1, value2 ->*
*.../ value3*

**Description**

A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed, the type of the results, and so on.

Runtime Exception

If any runtime exceptions can be thrown by the execution of an instruction, that instruction must not throw any runtime exceptions except for instances of System Exception.

**Notes**

Commands not strictly part of the specification of an instruction are set aside as notes at the end of the description.

Each cell in the instruction format diagram represents a single 8-bit byte. The instruction's mnemonic is its name. Its opcode is its numeric representation and is given in both decimal and hexadecimal forms. Only the numeric representation is actually present in the Java Card virtual machine code in a CAP file.

Keep in mind that there are "operands" generated at compile time and embedded within Java Card virtual machine instructions, as well as "operands" calculated at run time and supplied on the operand stack. Although they are supplied from several different areas, all these operands represent the same thing: values to be operated upon by the Java Card virtual machine instruction being executed. By implicitly taking many of its operands from its operand stack, rather than representing them explicitly in its compiled code as additional operand bytes, register numbers, etc., the Java Card virtual machine's code stays compact.

Some instructions are presented as members of a family of related instructions sharing a single description, format, and operand stack diagram. As such, a family of instructions includes several opcodes and opcode mnemonics; only the family mnemonic appears in the instruction format diagram, and a separate forms line lists all member mnemonics and opcodes. For example, the forms line for the sconst_<s> family of instructions, giving mnemonic and opcode information for the two instructions in that family (sconst_0 and sconst_1), is

Forms sconst_0 = 3 (0x3),
sconst_1 = 4 (0x4)

In the description of the Java Card virtual machine instructions, the effect of an instruction's execution on the operand stack (Section 3.5, "Frames" on page 3-3) of the current frame (Section 3.5, "Frames" on page 3-3) is represented textually, with the stack growing from left to right and each word represented separately. Thus,

Stack…, value1, value2 ->
…, result

shows an operation that begins by having a one-word value2 on top of the operand stack with a one-word value1 just beneath it. As a result of the execution of the instruction, value1 and value2 are popped from the operand stack and replaced by a one-word result, which has been calculated by the instruction. The remainder of the operand stack, represented by an ellipsis (…), is unaffected by the instruction's execution.

The type int takes two words on the operand stack. In the operand stack representation, each word is represented separately using a dot notation:

Stack…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

The Virtual Machine Specification for the Java Card Platform, v2.2.2 does not mandate how the two words are used to represent the 32-bit int value; it only requires that a particular implementation be internally consistent.

## 7.5.1 aaload

Load reference from array

Format

| |
|---|
| *aaload* |

Forms

aaload = 36 (0x24)

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 short. 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 top of the operand stack.

Runtime Exceptions

If arrayref is null, aaload throws a NullPointerException.

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

Notes

In some circumstances, the aaload instruction may throw a SecurityException if the current context (Section 3.4, "Contexts" on page 3-2) is not the owning context (Section 3.4, "Contexts" on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

## 7.5.2 aastore

Store into reference array

Format

| |
|---|
| *aastore* |

Forms

aastore = 55 (0x37)

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 short and the value must be of type reference. The arrayref, index and value are popped from the operand stack. The reference value is stored as the component of the array at index.

At runtime the type of value must be confirmed to be assignment compatible with the type of the components of the array referenced by arrayref. Assignment of a value of reference type S (source) to a variable of reference type T (target) is allowed only when the type S supports all of the operations defined on type T. The detailed rules follow:

- 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 interface type[1], then:
  - If T is a class type, then T must be Object (Section 2.2.2.4, "Classes" on page 2-7);
  - If T is an interface type, T must be the same interface as S or a superinterface of S.
- If S is an array type, namely the type SC[], that is, an array of components of type SC, then:
  - If T is a class type, then T must be Object.
  - If T is an array type, namely the type TC[], an array of components of type TC, then one of the following must be true:

    TC and SC are the same primitive type (Section 3.1, "Data Types and Values" on page 3-1").

    TC and SC are reference types[2] (Section 3.1, "Data Types and Values" on page 3-1) with type SC assignable to TC, by these rules.
- If T is an interface type, T must be one of the interfaces implemented by arrays.

Runtime 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 value is not assignment compatible with the actual type of the component of the array, aastore throws an ArrayStoreException.

Notes

1. When both *S* and *T* are arrays of reference types, this algorithm is applied recursively using the types of the arrays, namely *SC* and *TC*. In the recursive call, *S*, which was *SC* in the original call, may be an interface type. This rule can only be reached in this manner. Similarly, in the recursive call,*T*, which was *TC* in the original call, may be an interface type.
2. This version of the Java Card virtual machine does not support multi-dimensional arrays. Therefore, neither *SC* or *TC* can be an array type.

In some circumstances, the aastore instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

### 7.5.3 aconst_null

Push null

Format

|  |
|---|
| *aconst_null* |

Forms

aconst_null = 1 (0x1)

Stack

… ->
…, null

Description

Push the null object reference onto the operand stack.

### 7.5.4 aload

Load reference from local variable

Format

|  |
|---|
| *aload* |
| *index* |

Forms

aload = 21 (0x15)

Stack

… ->
…, objectref

Description

The index is an unsigned byte that must be a valid index into the local variables of the current frame (Section 3.5, “Frames” on page 3-3). 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 is intentional.

## 7.5.5 aload_<n>

Load reference from local variable

Format

| |
|---|
| *aload_<n>* |

Forms

aload_0 = 24 (0x18)
aload_1 = 25 (0x19)
aload_2 = 26 (0x1a)
aload_3 = 27 (0x1b)

Stack

… ->
…, objectref

Description

The <n> must be a valid index into the local variables of the current frame (Section 3.5, “Frames” on page 3-3). 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 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.

## 7.5.6 anewarray

Create new array of reference

Format

| |
|---|
| *anewarray* |
| *indexbyte1* |
| *indexbyte2* |

Forms

anewarray = 145 (0x91)

Stack

…, count ->
…, arrayref

Description

The count must be of type short. 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 constant pool of the current package (Section 3.5, "Frames" on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The item at that index in the constant pool must be of type CONSTANT_Classref (Section 6.7.1, "CONSTANT_Classref" on page 6-16), a reference to a class or interface type. The reference is resolved. A new array with components of that type, of length count, is allocated from the 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.

Runtime Exception

If count is less than zero, the anewarray instruction throws a NegativeArraySizeException.

## 7.5.7 areturn

Return reference from method

Format

| |
|---|
| *areturn* |

Forms

areturn = 119 (0x77)

Stack

…, objectref ->
[empty]

Description

The objectref must be of type reference. The objectref is popped from the operand stack of the current frame (Section 3.5, “Frames” on page 3-3) 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 virtual machine then reinstates the frame of the invoker and returns control to the invoker.

## 7.5.8 arraylength

Get length of array

Format

| |
|---|
| *arraylength* |

Forms

arraylength = 146 (0x92)

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 top of the operand stack as a short.

Runtime Exception

If arrayref is null, the arraylength instruction throws a NullPointerException.

Notes

In some circumstances, the arraylength instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

## 7.5.9 astore

Store reference into local variable

Format

| |
|---|
| *astore* |
| *index* |

Forms

astore = 40 (0x28)

Stack

…, objectref ->
…

Description

The index is an unsigned byte that must be a valid index into the local variables of the current frame (Section 3.5, “Frames” on page 3-3). The objectref on the top of the operand stack must be of type returnAddress or of type reference. The objectref 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 Java’s finally keyword. The aload instruction cannot be used to load a value of type returnAddress from a local variable onto the operand stack. This asymmetry with the astore instruction is intentional.

## 7.5.10 astore_<n>

Store reference into local variable

Format

*astore_<n>*

Forms

astore_0 = 43 (0x2b)
astore_1 = 44 (0x2c)
astore_2 = 45 (0x2d)
astore_3 = 46 (0x2e)

Stack

…, objectref ->
…

Description

The <n> must be a valid index into the local variables of the current frame (Section 3.5, "Frames" on page 3-3). 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 Java's finally keyword. An aload_<n> instruction cannot be used to load a value of type returnAddress from a local variable onto the operand stack. This asymmetry with the corresponding astore_<n> instruction is intentional.

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

## 7.5.11 athrow

Throw exception or error

Format

*athrow*

Forms

athrow = 147 (0x93)

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 frame (Section 3.5, “Frames” on page 3-3) for the most recent catch clause that catches the class of objectref or one of its superclasses.

If a catch clause 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 appropriate clause is found in the current frame, that frame is popped, the frame of its invoker is reinstated, and the objectref is rethrown.

If no catch clause is found that handles this exception, the virtual machine exits.

Runtime Exception

If objectref is null, athrow throws a NullPointerException instead of objectref.

Notes

In some circumstances, the athrow instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

## 7.5.12 baload

Load byte or boolean from array

Format

| |
|---|
| *baload* |

Forms

baload = 37 (0x25)

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 short. 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 a short value, and pushed onto the top of the operand stack.

Runtime Exceptions

If arrayref is null, baload throws a NullPointerException.

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

Notes

In some circumstances, the baload instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

## 7.5.13 bastore

Store into byte or boolean array

Format

| |
|---|
| *bastore* |

Forms

bastore = 56 (0x38)

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 value must both be of type short. The arrayref, index and value are popped from the operand stack. The short value is truncated to a byte and stored as the component of the array indexed by index.

Runtime Exceptions

If arrayref is null, bastore throws a NullPointerException.

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

Notes

In some circumstances, the bastore instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

## 7.5.14 bipush

Push byte

Format

| |
|---|
| *bipush* |
| *byte* |

Forms

bipush = 18 (0x12)

Stack

… ->
…, value.word1, value.word2

Description

The immediate byte is sign-extended to an int, and the resulting value is pushed onto the operand stack.

**Note –** If a virtual machine does not support the int data type, the bipush instruction will not be available.

## 7.5.15 bspush

Push byte

Format

| |
|---|
| *bspush* |
| *byte* |

Forms

bspush = 16 (0x10)

Stack

… ->
…, value

Description

The immediate byte is sign-extended to a short, and the resulting value is pushed onto the operand stack.

## 7.5.16 checkcast

Check whether object is of given type

Format

| |
|---|
| *checkcast* |
| *atype* |
| *indexbyte1* |
| *indexbyte2* |

Forms

checkcast = 148 (0x94)

Stack

…, objectref ->
…, objectref

Description

The unsigned byte atype is a code that indicates if the type against which the object is being checked is an array type or a class type. It must take one of the following values or zero:

**TABLE 7-2** Array Values

| Array Type | atype |
|---|---|
| `T_BOOLEAN` | `10` |
| `T_BYTE` | `11` |
| `T_SHORT` | `12` |
| `T_INT` | `13` |
| `T_REFERENCE` | `14` |

If the value of atype is 10, 11, 12, or 13, the values of the indexbyte1 and indexbyte2 must be zero, and the value of atype indicates the array type against which to check the object. Otherwise the unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, “Frames” on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The item at that index in the constant pool must be of type CONSTANT_Classref (Section 6.7.1, “CONSTANT_Classref” on page 6-16), a reference to a class or interface type. The reference is resolved. If the value of atype is 14, the object is checked against an array type that is an array of object references of the type of the resolved class. If the value of atype is zero, the object is checked against a class or interface type that is the resolved class.

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

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

- If S is 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 interface type[1], then:

1. When both *S* and *T* are arrays of reference types, this algorithm is applied recursively using the types of the arrays, namely *SC* and *TC*. In the recursive call, *S*, which was *SC* in the original call, may be an interface type. This rule can only be reached in this manner. Similarly, in the recursive call,*T*, which was *TC* in the original call, may be an interface type.

  - If T is a class type, then T must be Object (Section 2.2.2.4, “Classes” on page 2-7);
  - If T is an interface type, T must be the same interface as S or a superinterface of S.
- If S is an array type, namely the type SC[], that is, an array of components of type SC, then:
  - If T is a class type, then T must be Object.
  - If T is an array type, namely the type TC[], an array of components of type TC, then one of the following must be true:

    TC and SC are the same primitive type (Section 3.1, “Data Types and Values” on page 3-1).

    TC and SC are reference types[1] (Section 3.1, “Data Types and Values” on page 3-1) with type SC assignable to TC, by these rules.
  - If T is an interface type, T must be one of the interfaces implemented by arrays.

Runtime Exception

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

Notes

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

In some circumstances, the checkcast instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the value of atype may not be 13 (array type = T_INT).

## 7.5.17 dup

Duplicate top operand stack word

1. This version of the Java Card virtual machine specification does not support multi-dimensional arrays. Therefore, neither *SC* or *TC* can be an array type.

Format

| *dup* |
|---|

Forms

dup = 61 (0x3d)

Stack

…, word ->
…, word, word

Description

The top word on the operand stack is duplicated and pushed onto the operand stack.

The dup instruction must not be used unless word contains a 16-bit data type.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the dup instruction operates on an untyped word, ignoring the type of data it contains.

## 7.5.18 dup_x

Duplicate top operand stack words and insert below

Format

| *dup_x* |
|---|
| *mn* |

Forms

dup_x = 63 (0x3f)

Stack

…, wordN, …, wordM, …, word1 ->
…, wordM, …, word1, wordN, …, wordM, …, word1

Description

The unsigned byte mn is used to construct two parameter values. The high nibble, (mn & 0xf0) >> 4, is used as the value m. The low nibble, (mn & 0xf), is used as the value n. Permissible values for m are 1 through 4. Permissible values for n are 0 and m through m+4.

For positive values of n, the top m words on the operand stack are duplicated and the copied words are inserted n words down in the operand stack. When n equals 0, the top m words are copied and placed on top of the stack.

The dup_x instruction must not be used unless the ranges of words 1 through m and words m+1 through n each contain either a 16-bit data type, two 16-bit data types, a 32-bit data type, a 16-bit data type and a 32-bit data type (in either order), or two 32-bit data types.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the dup_x instruction operates on untyped words, ignoring the types of data they contain.

If a virtual machine does not support the int data type, the permissible values for m are 1 or 2, and permissible values for n are 0 and m through m+2.

## 7.5.19 dup2

Duplicate top two operand stack words

Format

| |
|---|
| *dup2* |

Forms

dup2 = 62 (0x3e)

Stack

…, word2, word1 ->
…, word2, word1, word2, word1

Description

The top two words on the operand stack are duplicated and pushed onto the operand stack, in the original order.

The dup2 instruction must not be used unless each of word1 and word2 is a word that contains a 16-bit data type or both together are the two words of a single 32-bit datum.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the dup2 instruction operates on untyped words, ignoring the types of data they contain.

## 7.5.20 getfield_<t>

Fetch field from object

Format

|  |
|---|
| *getfield_<t>* |
| *index* |

Forms

getfield_a = 131 (0x83)
getfield_b = 132 (0x84)
getfield_s = 133 (0x85)
getfield_i = 134 (0x86)

Stack

…, objectref ->
…, value

OR

…, objectref ->
…, value.word1, value.word2

Description

The objectref, which must be of type reference, is popped from the operand stack. The unsigned index is used as an index into the constant pool of the current package (Section 3.5, "Frames" on page 3-3). The constant pool item at the index must be of type CONSTANT_InstanceFieldref (Section 6.7.2, "CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref" on page 6-18), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches t, as follows:

- a field must be of type reference

- b field must be of type byte or type boolean
- s field must be of type short
- i field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset[1]. The value at that offset into the class instance referenced by objectref is fetched. If the value is of type byte or type boolean, it is sign-extended to a short. The value is pushed onto the operand stack.

Runtime Exception

If objectref is null, the getfield_<t> instruction throws a NullPointerException.

Notes

In some circumstances, the getfield_<t> instruction may throw a SecurityException if the current context (Section 3.4, "Contexts" on page 3-2) is not the owning context (Section 3.4, "Contexts" on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the getfield_i instruction will not be available.

### 7.5.21 getfield_<t>_this

Fetch field from current object

Format

| |
|---|
| *getfield_<t>_this* |
| *index* |

Forms

getfield_a_this = 173 (0xad)
getfield_b_this = 174 (0xae)
getfield_s_this = 175 (0xaf)
getfield_i_this = 176 (0xb0)

Stack

1. The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.

… ->
…, value

OR

… ->
…, value.word1, value.word2

Description

The currently executing method must be an instance method. The local variable at index 0 must contain a reference objectref to the currently executing method's this parameter. The unsigned index is used as an index into the constant pool of the current package (Section 3.5, "Frames" on page 3-3). The constant pool item at the index must be of type CONSTANT_InstanceFieldref (Section 6.7.2, "CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref" on page 6-18), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches t, as follows:

- a field must be of type reference
- b field must be of type byte or type boolean
- s field must be of type short
- i field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset[1]. The value at that offset into the class instance referenced by objectref is fetched. If the value is of type byte or type boolean, it is sign-extended to a short. The value is pushed onto the operand stack.

Runtime Exception

If objectref is null, the getfield_<t>_this instruction throws a NullPointerException.

Notes

1. The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.

In some circumstances, the getfield_<t>_this instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the getfield_i_this instruction will not be available.

## 7.5.22 getfield_<t>_w

Fetch field from object (wide index)

Format

| |
|---|
| *getfield_<t>_w* |
| *indexbyte1* |
| *indexbyte2* |

Forms

getfield_a_w = 169 (0xa9)
getfield_b_w = 170 (0xaa)
getfield_s_w = 171 (0xab)
getfield_i_w = 172 (0xac)

Stack

…, objectref ->
…, value

OR

…, objectref ->
…, value.word1, value.word2

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 constant pool of the current package (Section 3.5, “Frames” on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at the index must be of type CONSTANT_InstanceFieldref (Section 6.7.2, “CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref” on page 6-18), a reference to a class and a field token. The item must resolve to a field of type reference.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches t, as follows:

- a field must be of type reference
- b field must be of type byte or type boolean
- s field must be of type short
- i field must be of type int

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset[1]. The value at that offset into the class instance referenced by objectref is fetched. If the value is of type byte or type boolean, it is sign-extended to a short. The value is pushed onto the operand stack.

Runtime Exception

If objectref is null, the getfield_<t>_w instruction throws a NullPointerException.

Notes

In some circumstances, the getfield_<t>_w instruction may throw a SecurityException if the current context (Section 3.4, "Contexts" on page 3-2) is not the owning context Section 3.4, "Contexts" on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the getfield_i_w instruction will not be available.

## 7.5.23 getstatic_<t>

Get static field from class

Format

| |
|---|
| *getstatic_<t>* |
| *indexbyte1* |
| *indexbyte2* |

1. The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.

Forms

getstatic_a = 123 (0x7b)
getstatic_b = 124 (0x7c)
getstatic_s = 125 (0x7d)
getstatic_i = 126 (0x7e)

Stack

… ->
…, value

OR

… ->
…, value.word1, value.word2

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, "Frames" on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at the index must be of type CONSTANT_StaticFieldref (Section 6.7.3, "CONSTANT_StaticFieldref and CONSTANT_StaticMethodref" on page 6-19), a reference to a static field.

The item must resolve to a field with a type that matches t, as follows:

- a field must be of type reference
- b field must be of type byte or type boolean
- s field must be of type short
- i field must be of type int

The width of a class field is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The item is resolved, determining the class field. The value of the class field is fetched. If the value is of type byte or boolean, it is sign-extended to a short. The value is pushed onto the operand stack.

Notes

If a virtual machine does not support the int data type, the getstatic_i instruction will not be available.

### 7.5.24 goto

Branch always

Format

|  |
|---|
| *goto* |
| *branch* |

Forms

goto = 112 (0x70)

Stack

No change

Description

The value branch is used as a signed 8-bit offset. 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.

## 7.5.25 goto_w

Branch always (wide index)

Format

|  |
|---|
| *goto_w* |
| *branchbyte1* |
| *branchbyte2* |

Forms

goto_w = 168 (0xa8)

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.

## 7.5.26 i2b

Convert int to byte

Format

| |
|---|
| *i2b* |

Forms

i2b = 93 (0x5d)

Stack

…, value.word1, value.word2 ->
…, result

Description

The value on top of the operand stack must be of type int. It is popped from the operand stack and converted to a byte result by taking the low-order 16 bits of the int value, and discarding the high-order 16 bits. The low-order word is truncated to a byte, then sign-extended to a short result. The result is pushed onto the operand stack.

Notes

The i2b instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of value. The result may also not have the same sign as value.

If a virtual machine does not support the int data type, the i2b instruction will not be available.

## 7.5.27 i2s

Convert int to short

Format

| |
|---|
| *i2s* |

Forms

i2s = 94 (0x5e)

Stack

…, value.word1, value.word2 ->
…, result

Description

The value on top of the operand stack must be of type int. It is popped from the operand stack and converted to a short result by taking the low-order 16 bits of the int value and discarding the high-order 16 bits. The result is pushed onto the operand stack.

Notes

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

If a virtual machine does not support the int data type, the i2s instruction will not be available.

## 7.5.28 iadd

Add int

Format

| |
|---|
| *iadd* |

Forms

iadd = 66 (0x42)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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.

If an iadd instruction overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide two's-complement format. 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.

Notes

If a virtual machine does not support the int data type, the iadd instruction will not be available.

## 7.5.29 iaload

Load int from array

Format

| |
|---|
| *iaload* |

Forms

iaload = 39 (0x27)

Stack

…, arrayref, index ->
…, value.word1, value.word2

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 short. 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 top of the operand stack.

Runtime Exceptions

If arrayref is null, iaload throws a NullPointerException.

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

Notes

In some circumstances, the iaload instruction may throw a SecurityException if the current context Section 3.4, "Contexts" on page 3-2) is not the owning context (Section 3.4, "Contexts" on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the iaload instruction will not be available.

## 7.5.30 iand

Boolean AND int

Format

| |
|---|
| *iand* |

Forms

iand = 84 (0x54)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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.

Notes

If a virtual machine does not support the int data type, the iand instruction will not be available.

## 7.5.31 iastore

Store into int array

Format

| |
|---|
| *iastore* |

Forms

iastore = 58 (0x3a)

Stack

…, arrayref, index, value.word1, value.word2 ->
…

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 short and value must be of type int. The arrayref, index and value are popped from the operand stack. The int value is stored as the component of the array indexed by index.

Runtime Exception

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.

Notes

In some circumstances, the iastore instruction may throw a SecurityException if the current context (Section 3.4, "Contexts" on page 3-2) is not the owning context (Section 3.4, "Contexts" on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the iastore instruction will not be available.

## 7.5.32 icmp

Compare int

Format

| |
|---|
| *icmp* |

Forms

icmp = 95 (0x5f)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result

Description

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

Notes

If a virtual machine does not support the int data type, the icmp instruction will not be available.

## 7.5.33 iconst_<i>

Push int constant

Format

| *iconst_<i>* |
|---|

Forms

iconst_m1 = 10 (0x09)
iconst_0 = 11 (0xa)
iconst_1 = 12 (0xb)
iconst_2 = 13 (0xc)
iconst_3 = 14 (0xd)
iconst_4 = 15 (0xe)
iconst_5 = 16 (0xf)

Stack

… ->
…, <i>.word1, <i>.word2

Description

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

Notes

If a virtual machine does not support the int data type, the iconst_<i> instruction will not be available.

## 7.5.34 idiv

Divide int

Format

| *idiv* |
|---|

Forms

idiv = 72 (0x48)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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 expression value1 / value2. The result is pushed onto the operand stack.

An int division rounds towards 0; that is, the quotient produced for int values in n/d is an int value q whose magnitude is as large as possible while satisfying | d · q | <= | n |. Moreover, q is a positive when | n | >= | d | and n and d have the same sign, but q is negative when | n | >= | 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 the largest possible magnitude for the int type, and the divisor is –1, then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case.

Runtime Exception

If the value of the divisor in an int division is 0, idiv throws an ArithmeticException.

Notes

If a virtual machine does not support the int data type, the idiv instruction will not be available.

## 7.5.35 if_acmp<cond>

Branch if reference comparison succeeds.

Format

| |
|---|
| *if_acmp<cond>* |
| *branch* |

Forms

if_acmpeq = 104 (0x68)
if_acmpne = 105 (0x69)

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 comparisons are as follows:

- eq succeeds if and only if value1 = value2
- ne succeeds if and only if value1 ¼ value2

If the comparison succeeds, branch is used as signed 8-bit offset, and execution 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, execution proceeds at the address of the instruction following this if_acmp<cond> instruction.

### 7.5.36 if_acmp<cond>_w

Branch if reference comparison succeeds (wide index)

Format

| |
|---|
| *if_acmp<cond>_w* |
| *branchbyte1* |
| *branchbyte2* |

Forms

if_acmpeq_w = 160 (0xa0)
if_acmpne_w = 161 (0xa1)

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 comparisons are as follows:

- eq succeeds if and only if value1 = value2
- ne succeeds if and only if value1 ¼ value2

If the comparison succeeds, 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 if_acmp<cond>_w instruction. The target address must be that of an opcode of an instruction within the method that contains this if_acmp<cond>_w instruction.

Otherwise, execution proceeds at the address of the instruction following this if_acmp<cond>_w instruction.

## 7.5.37 if_scmp<cond>

Branch if short comparison succeeds

Format

|  |
|---|
| *if_scmp<cond>* |
| *branch* |

Forms

if_scmpeq = 106 (0x6a)
if_scmpne = 107 (0x6b)
if_scmplt = 108 (0x6c)
if_scmpge = 109 (0x6d)
if_scmpgt = 110 (0x6e)
if_scmple = 111 (0x6f)

Stack

…, value1, value2 ->
…

Description

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

- eq succeeds if and only if value1 = value2
- ne succeeds if and only if value1 ¼ value2
- lt succeeds if and only if value1 < value2
- le succeeds if and only if value1 £ value2
- gt succeeds if and only if value1 > value2
- ge succeeds if and only if value1 Š value2

If the comparison succeeds, branch is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this if_scmp<cond> instruction. The target address must be that of an opcode of an instruction within the method that contains this if_scmp<cond> instruction.

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

## 7.5.38 if_scmp<cond>_w

Branch if short comparison succeeds (wide index)

Format

| |
|---|
| *if_scmp<cond>_w* |
| *branchbyte1* |
| *branchbyte2* |

Forms

if_scmpeq_w = 162 (0xa2)
if_scmpne_w = 163 (0xa3)
if_scmplt_w = 164 (0xa4)
if_scmpge_w = 165 (0xa5)
if_scmpgt_w = 166 (0xa6)
if_scmple_w = 167 (0xa7)

Stack

…, value1, value2 ->
…

Description

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

- eq succeeds if and only if value1 = value2
- ne succeeds if and only if value1 ¼ value2
- lt succeeds if and only if value1 < value2
- le succeeds if and only if value1 £ value2
- gt succeeds if and only if value1 > value2
- ge succeeds if and only if value1 Š value2

If the comparison succeeds, 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 if_scmp<cond>_w instruction. The target address must be that of an opcode of an instruction within the method that contains this if_scmp<cond>_w instruction.

Otherwise, execution proceeds at the address of the instruction following this if_scmp<cond>_w instruction.

## 7.5.39 if<cond>

Branch if short comparison with zero succeeds

Format

| |
|---|
| *if<cond>* |
| *branch* |

Forms

ifeq = 96 (0x60)
ifne = 97 (0x61)
iflt = 98 (0x62)
ifge = 99 (0x63)
ifgt = 100 (0x64)
ifle = 101 (0x65)

Stack

…, value ->
…

Description

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

- eq succeeds if and only if value = 0
- ne succeeds if and only if value ¼ 0
- lt succeeds if and only if value < 0
- le succeeds if and only if value £ 0
- gt succeeds if and only if value > 0
- ge succeeds if and only if value Š 0

If the comparison succeeds, branch is used as signed 8-bit offset, and execution 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.

## 7.5.40 if<cond>_w

Branch if short comparison with zero succeeds (wide index)

Format

| |
|---|
| *if<cond>_w* |
| *branchbyte1* |
| *branchbyte2* |

Forms

ifeq_w = 152 (0x98)
ifne_w = 153 (0x99)
iflt_w = 154 (0x9a)
ifge_w = 155 (0x9b)
ifgt_w = 156 (0x9c)
ifle_w = 157 (0x9d)

Stack

…, value ->
…

Description

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

- eq succeeds if and only if value = 0
- ne succeeds if and only if value ¼ 0
- lt succeeds if and only if value < 0
- le succeeds if and only if value £ 0
- gt succeeds if and only if value > 0
- ge succeeds if and only if value Š 0

If the comparison succeeds, 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 if<cond>_w instruction. The target address must be that of an opcode of an instruction within the method that contains this if<cond>_w instruction.

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

## 7.5.41 ifnonnull

Branch if reference not null

Format

| |
|---|
| *ifnonnull* |
| *branch* |

Forms

ifnonnull = 103 (0x67)

Stack

…, value ->
…

Description

The value must be of type reference. It is popped from the operand stack. If the value is not null, branch is used as signed 8-bit offset, and execution 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.

## 7.5.42 ifnonnull_w

Branch if reference not null (wide index)

Format

| |
|---|
| *ifnonnull_w* |
| *branchbyte1* |
| *branchbyte2* |

Forms

ifnonnull_w = 159 (0x9f)

Stack

…, value ->
…

Description

The value must be of type reference. It is popped from the operand stack. If the value is not null, 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 ifnonnull_w instruction. The target address must be that of an opcode of an instruction within the method that contains this ifnonnull_w instruction.

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

## 7.5.43 ifnull

Branch if reference is null

Format

| |
|---|
| *ifnull* |
| *branch* |

Forms

ifnull = 102 (0x66)

Stack

…, value ->
…

Description

The value must be of type reference. It is popped from the operand stack. If the value is null, branch is used as signed 8-bit offset, and execution 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.

## 7.5.44 ifnull_w

Branch if reference is null (wide index)

Format

| |
|---|
| *ifnull_w* |
| *branchbyte1* |
| *branchbyte2* |

Forms

ifnull_w = 158 (0x9e)

Stack

…, value ->
…

Description

The value must be of type reference. It is popped from the operand stack. If the value is null, 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 ifnull_w instruction. The target address must be that of an opcode of an instruction within the method that contains this ifnull_w instruction.

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

## 7.5.45 iinc

Increment local int variable by constant

Format

| |
|---|
| *iinc* |
| *index* |
| *const* |

Forms

iinc = 90 (0x5a)

Stack

No change

Description

The index is an unsigned byte. Both index and index + 1 must be valid indices into the local variables of the current frame (Section 3.5, "Frames" on page 3-3). The local variables at index and index + 1 together must contain an int. The const is an immediate signed byte. The value const is first sign-extended to an int, then the int contained in the local variables at index and index + 1 is incremented by that amount.

Notes

If a virtual machine does not support the int data type, the iinc instruction will not be available.

## 7.5.46 iinc_w

Increment local int variable by constant

Format

| |
|---|
| *iinc_w* |
| *index* |
| *byte1* |
| *byte2* |

Forms

iinc_w = 151 (0x97)

Stack

No change

Description

The index is an unsigned byte. Both index and index + 1 must be valid indices into the local variables of the current frame (Section 3.5, “Frames” on page 3-3). The local variables at index and index + 1 together must contain an int. 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 const. The int contained in the local variables at index and index + 1 is incremented by const.

Notes

If a virtual machine does not support the int data type, the iinc_w instruction will not be available.

## 7.5.47 iipush

Push int

Format

| |
|---|
| *iipush* |
| *byte1* |
| *byte2* |
| *byte3* |
| *byte4* |

Forms

iipush = 20 (0x14)

Stack

… ->
…, value1.word1, value1.word2

Description

The immediate unsigned byte1, byte2, byte3, and byte4 values are assembled into a signed int where the value of the int is (byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4. The resulting value is pushed onto the operand stack.

Notes

If a virtual machine does not support the int data type, the iipush instruction will not be available.

## 7.5.48 iload

Load int from local variable

Format

| |
|---|
| *iload* |
| *index* |

Forms

iload = 23 (0x17)

Stack

… ->
…, value1.word1, value1.word2

Description

The index is an unsigned byte. Both index and index + 1 must be valid indices into the local variables of the current frame (Section 3.5, “Frames” on page 3-3). The local variables at index and index + 1 together must contain an int. The value of the local variables at index and index + 1 is pushed onto the operand stack.

Notes

If a virtual machine does not support the int data type, the iload instruction will not be available.

## 7.5.49 iload_<n>

Load int from local variable

Format

| |
|---|
| *iload_<n>* |

Forms

iload_0 = 32 (0x20)
iload_1 = 33 (0x21)
iload_2 = 34 (0x22)
iload_3 = 35 (0x23)

Stack

… ->
…, value1.word1, value1.word2

Description

Both <n> and <n> + 1 must be a valid indices into the local variables of the current frame (Section 3.5, "Frames" on page 3-3). The local variables at <n> and <n> + 1 together must contain an int. The value of the local variables at <n> and <n> + 1 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.

If a virtual machine does not support the int data type, the iload_<n> instruction will not be available.

## 7.5.50 ilookupswitch

Access jump table by key match and jump

Format

| |
|---|
| *ilookupswitch* |
| *defaultbyte1* |
| *defaultbyte2* |
| *npairs1* |
| *npairs2* |
| *match-offset pairs…* |

Pair Format

| |
|---|
| *matchbyte1* |
| *matchbyte2* |
| *matchbyte3* |

|  |
|---|
| *matchbyte4* |
| *offsetbyte1* |
| *offsetbyte2* |

Forms

ilookupswitch = 118 (0x76)

Stack

…, key.word1, key.word2 ->
…

Description

An ilookupswitch instruction is a variable-length instruction. Immediately after the ilookupswitch opcode follow a signed 16-bit value default, an unsigned 16-bit value npairs, and then npairs pairs. Each pair consists of an int match and a signed 16-bit offset. Each match is constructed from four unsigned bytes as (matchbyte1 << 24) | (matchbyte2 << 16) | (matchbyte3 << 8) | matchbyte4. Each offset is constructed from two unsigned bytes as (offsetbyte1 << 8) | offsetbyte2.

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

The key must be of type int and is popped from the operand stack and 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 ilookupswitch 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 ilookupswitch 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 ilookupswitch instruction.

Notes

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

If a virtual machine does not support the int data type, the ilookupswitch instruction will not be available.

## 7.5.51 imul

Multiply int

Format

| |
|---|
| *imul* |

Forms

imul = 70 (0x46)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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.

If an imul instruction overflows, then the result is the low-order bits of the mathematical product as an int. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two values.

Notes

If a virtual machine does not support the int data type, the imul instruction will not be available.

## 7.5.52 ineg

Negate int

Format

| |
|---|
| *ineg* |

Forms

ineg = 76 (0x4c)

Stack

…, value.word1, value.word2 ->
…, result.word1, result.word2

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 Card 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.

Notes

If a virtual machine does not support the int data type, the ineg instruction will not be available.

## 7.5.53 instanceof

Determine if object is of given type

Format

| |
|---|
| *instanceof* |
| *atype* |
| *indexbyte1* |
| *indexbyte2* |

Forms

instanceof = 149 (0x95)

Stack

…, objectref ->
…, result

Description

The unsigned byte atype is a code that indicates if the type against which the object is being checked is an array type or a class type. It must take one of the following values or zero:

**TABLE 7-3** Array Values

| Array Type | atype |
|---|---|
| `T_BOOLEAN` | `10` |
| `T_BYTE` | `11` |
| `T_SHORT` | `12` |
| `T_INT` | `13` |
| `T_REFERENCE` | `14` |

If the value of atype is 10, 11, 12, or 13, the values of the indexbyte1 and indexbyte2 must be zero, and the value of atype indicates the array type against which to check the object. Otherwise the unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, "Frames" on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The item at that index in the constant pool must be of type CONSTANT_Classref (Section 6.7.1, "CONSTANT_Classref" on page 6-16), a reference to a class or interface type. The reference is resolved. If the value of atype is 14, the object is checked against an array type that is an array of object references of the type of the resolved class. If the value of atype is zero, the object is checked against a class or interface type that is the resolved class.

The objectref must be of type reference. It is popped from the operand stack. If objectref is not null and is an instance of the resolved class, array or interface, the instanceof instruction pushes a short result of 1 on the operand stack. Otherwise it pushes a short result of 0.

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

- If S is 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 interface type[1], then:

1. When both *S* and *T* are arrays of reference types, this algorithm is applied recursively using the types of the arrays, namely *SC* and *TC*. In the recursive call, *S*, which was *SC* in the original call, may be an interface type. This rule can only be reached in this manner. Similarly, in the recursive call,*T*, which was *TC* in the original call, may be an interface type.

  - If T is a class type, then T must be Object (Section 2.2.2.4, “Classes” on page 2-7);
  - If T is an interface type, T must be the same interface as S or a superinterface of S.
- If S is an array type, namely the type SC[], that is, an array of components of type SC, then:
  - If T is a class type, then T must be Object.
  - If T is an array type, namely the type TC[], an array of components of type TC, then one of the following must be true:

    TC and SC are the same primitive type (Section 3.1, “Data Types and Values” on page 3-1).

    TC and SC are reference types[1] (Section 3.1, “Data Types and Values” on page 3-1) with type SC assignable to TC, by these rules.
  - If T is an interface type, T must be one of the interfaces implemented by arrays.

Notes

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

In some circumstances, the instanceof instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the value of atype may not be 13 (array type = T_INT).

## 7.5.54 invokeinterface

Invoke interface method

Format

| |
|---|
| *invokeinterface* |
| *nargs* |

1. This version of the Java Card virtual machine specification does not support multi-dimensional arrays. Therefore, neither *SC* or *TC* can be an array type.

| |
|---|
| *indexbyte1* |
| *indexbyte2* |
| *method* |

Forms

invokeinterface = 142 (0x8e)

Stack

…, objectref, [arg1, [arg2 …]] ->
…

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, “Frames” on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at that index must be of type CONSTANT_Classref (Section 6.7.1, “CONSTANT_Classref” on page 6-16), a reference to an interface class. The specified interface is resolved.

The nargs operand is an unsigned byte that must not be zero.

The method operand is an unsigned byte that is the interface method token for the method to be invoked. The interface method must not be <init> or an instance initialization method.

The objectref must be of type reference and must be followed on the operand stack by nargs – 1 words of arguments. The number of words of arguments and the type and order of the values they represent must be consistent with those of the selected interface method.

The interface table of the class of the type of objectref is determined. If objectref is an array type, then the interface table of class Object (Section 2.2.2.4, “Classes” on page 2-7) is used. The interface table is searched for the resolved interface. The result of the search is a table that is used to map the method token to a index.

The index is an unsigned byte that is used as an index into the method table of the class of the type of objectref. If the objectref is an array type, then the method table of class Object is used. The table entry at that index includes a direct reference to the method’s code and modifier information.

The nargs – 1 words of arguments and objectref are popped from the operand stack. A new stack frame is created for the method being invoked, and objectref and the arguments are made the values of its first nargs words of local variables, with objectref in local variable 0, arg1 in local variable 1, and so on. The new stack frame is then made current, and the Java Card 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.

Runtime Exception

If objectref is null, the invokeinterface instruction throws a NullPointerException.

Notes

In some circumstances, the invokeinterface instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the context (Section 3.4, “Contexts” on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*. If the current context is not the object’s context and the Java Card RE permits invocation of the method, the invokeinterface instruction will cause a context switch (Section 3.4, “Contexts” on page 3-2) to the object’s context before invoking the method, and will cause a return context switch to the previous context when the invoked method returns.

## 7.5.55 invokespecial

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

Format

| |
|---|
| *invokespecial* |
| *indexbyte1* |
| *indexbyte2* |

Forms

invokespecial = 140 (0x8c)

Stack

…, objectref, [arg1, [arg2 …]] ->
…

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, “Frames” on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. If the invoked method is a private instance method or an instance initialization method, the constant pool item at index must be of type CONSTANT_StaticMethodref (Section 6.7.3, "CONSTANT_StaticFieldref and CONSTANT_StaticMethodref" on page 6-19), a reference to a statically linked instance method. If the invoked method is a superclass method, the constant pool item at index must be of type CONSTANT_SuperMethodref (Section 6.7.2, "CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref" on page 6-18), a reference to an instance method of a specified class. The reference is resolved. The resolved method must not be <clinit>, a class or interface initialization method. If the method is <init>, an instance initialization method, then the method must only be invoked once on an uninitialized object, and before the first backward branch following the execution of the new instruction that allocated the object. Finally, if the resolved method is protected, and it is a member of a superclass of the current class, and the method is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class.

The resolved method includes the code for the method, an unsigned byte nargs that must not be zero, and the method's modifier information.

The objectref must be of type reference, and must be followed on the operand stack by nargs – 1 words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the selected instance method.

The nargs – 1 words of arguments and objectref are popped from the operand stack. A new stack frame is created for the method being invoked, and objectref and the arguments are made the values of its first nargs words of local variables, with objectref in local variable 0, arg1 in local variable 1, and so on. The new stack frame is then made current, and the Java Card 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.

Runtime Exception

If objectref is null, the invokespecial instruction throws a NullPointerException.

## 7.5.56 invokestatic

Invoke a class (static) method

Format

| |
|---|
| *invokestatic* |
| *indexbyte1* |
| *indexbyte2* |

Forms

invokestatic = 141 (0x8d)

Stack

…, [arg1, [arg2 …]] ->
…

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, "Frames" on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at that index must be of type CONSTANT_StaticMethodref (Section 6.7.3, "CONSTANT_StaticFieldref and CONSTANT_StaticMethodref" on page 6-19), a reference to a static method. The method must not be <init>, an instance initialization method, or <clinit>, a class or interface initialization method. It must be static, and therefore cannot be abstract.

The resolved method includes the code for the method, an unsigned byte nargs that may be zero, and the method's modifier information.

The operand stack must contain nargs words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the resolved method.

The nargs words of arguments are popped from the operand stack. A new stack frame is created for the method being invoked, and the words of arguments are made the values of its first nargs words of local variables, with arg1 in local variable 0, arg2 in local variable 1, and so on. The new stack frame is then made current, and the Java Card 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.

## 7.5.57 invokevirtual

Invoke instance method; dispatch based on class

Format

| |
|---|
| *invokevirtual* |
| *indexbyte1* |
| *indexbyte2* |

Forms

invokevirtual = 139 (0x8b)

Stack

…, objectref, [arg1, [arg2 …]] ->
…

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, “Frames” on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at that index must be of type CONSTANT_VirtualMethodref (Section 6.7.2, “CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref” on page 6-18), a reference to a class and a virtual method token. The specified method is resolved. The method must not be <init>, an instance initialization method, or <clinit>, a class or interface initialization method. Finally, if the resolved method is protected, and it is a member of a superclass of the current class, and the method is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class.

The resolved method reference includes an unsigned index into the method table of the resolved class and an unsigned byte nargs that must not be zero.

The objectref must be of type reference. The index is an unsigned byte that is used as an index into the method table of the class of the type of objectref. If the objectref is an array type, then the method table of class Object (Section 2.2.2.4, “Classes” on page 2-7) is used. The table entry at that index includes a direct reference to the method’s code and modifier information.

The objectref must be followed on the operand stack by nargs – 1 words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the selected instance method.

The nargs – 1 words of arguments and objectref are popped from the operand stack. A new stack frame is created for the method being invoked, and objectref and the arguments are made the values of its first nargs words of local variables, with objectref in local variable 0, arg1 in local variable 1, and so on. The new stack frame is then made current, and the Java Card 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.

Runtime Exception

If objectref is null, the invokevirtual instruction throws a NullPointerException.

In some circumstances, the invokevirtual instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the context (Section 3.4, “Contexts” on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*. If the current context is not the object's context and the Java Card RE permits invocation of the method, the invokevirtual instruction will cause a context switch (Section 3.4, "Contexts" on page 3-2) to the object's context before invoking the method, and will cause a return context switch to the previous context when the invoked method returns.

## 7.5.58 ior

Boolean OR int

Format

| |
|---|
| *ior* |

Forms

ior = 86 (0x56)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

Description

Both value1 and value2 must be of type int. The values 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.

Notes

If a virtual machine does not support the int data type, the ior instruction will not be available.

## 7.5.59 irem

Remainder int

Format

| |
|---|
| *irem* |

Forms

irem = 74 (0x4a)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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 expression 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 that the dividend is the negative int of largest possible magnitude for its type and the divisor is –1 (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

Runtime Exception

If the value of the divisor for a short remainder operator is 0, irem throws an ArithmeticException.

Notes

If a virtual machine does not support the int data type, the irem instruction will not be available.

## 7.5.60 ireturn

Return int from method

Format

| |
|---|
| *ireturn* |

Forms

ireturn = 121 (0x79)

Stack

…, value.word1, value.word2 ->
[empty]

Description

The value must be of type int. It is popped from the operand stack of the current frame (Section 3.5, “Frames” on page 3-3) 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 virtual machine then reinstates the frame of the invoker and returns control to the invoker.

Notes

If a virtual machine does not support the int data type, the ireturn instruction will not be available.

## 7.5.61 ishl

Shift left int

Format

| |
|---|
| *ishl* |

Forms

ishl = 78 (0x4e)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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 five 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.

If a virtual machine does not support the int data type, the ishl instruction will not be available.

## 7.5.62 ishr

Arithmetic shift right int

Format

| |
|---|
| *ishr* |

Forms

ishr = 80 (0x50)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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 five bits of value2. The result is pushed onto the operand stack.

Notes

The resulting value is Î(value1) / 2s°, where s is value2 & 0x1f. For nonnegative value1, this is equivalent (even if overflow occurs) 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.

Notes

If a virtual machine does not support the int data type, the ishr instruction will not be available.

## 7.5.63 istore

Store int into local variable

Format

| |
|---|
| *istore* |
| *index* |

Forms

istore = 42 (0x2a)

Stack

…, value.word1, value.word2 ->
…

Description

The index is an unsigned byte. Both index and index + 1 must be a valid index into the local variables of the current frame (Section 3.5, "Frames" on page 3-3). The value on top of the operand stack must be of type int. It is popped from the operand stack, and the local variables at index and index + 1 are set to value.

Notes

If a virtual machine does not support the int data type, the istore instruction will not be available.

## 7.5.64 istore_<n>

Store int into local variable

Format

| |
|---|
| *istore_<n>* |

Forms

istore_0 = 51 (0x33)
istore_1 = 52 (0x34)
istore_2 = 53 (0x35)
istore_3 = 54 (0x36)

Stack

…, value.word1, value.word2 ->
…

Description

Both <n> and <n> + 1 must be a valid indices into the local variables of the current frame (Section 3.5, "Frames" on page 3-3). The value on top of the operand stack must be of type int. It is popped from the operand stack, and the local variables at index and index + 1 are set to value.

Notes

If a virtual machine does not support the int data type, the istore_<n> instruction will not be available.

## 7.5.65 isub

Subtract int

Format

| *isub* |
|---|

Forms

isub = 68 (0x44)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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 zeros is the same as negation.

Despite the fact that overflow or underflow may occur, in which case the result may have a different sign than the true mathematical result, execution of an isub instruction never throws a runtime exception.

Notes

If a virtual machine does not support the int data type, the isub instruction will not be available.

## 7.5.66 itableswitch

Access jump table by int index and jump

Format

| |
|---|
| *itableswitch* |
| *defaultbyte1* |
| *defaultbyte2* |
| *lowbyte1* |
| *lowbyte2* |
| *lowbyte3* |
| *lowbyte4* |
| *highbyte1* |
| *highbyte2* |
| *highbyte3* |
| *highbyte4* |
| *jump offsets…* |

Offset Format

| |
|---|
| *offsetbyte1* |
| *offsetbyte2* |

Forms

itableswitch = 116 (0x74)

Stack

…, index ->
…

Description

An itableswitch instruction is a variable-length instruction. Immediately after the itableswitch opcode follow a signed 16-bit value default, a signed 32-bit value low, a signed 32-bit value high, and then high – low + 1 further signed 16-bit offsets. The value low must be less than or equal to high. The high – low + 1 signed 16-bit offsets are treated as a 0-based jump table. Each of the signed 16-bit values is constructed from two unsigned bytes as (byte1 << 8) | byte2. Each of the signed 32-bit values is constructed from four unsigned bytes as (byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4.

The index must be of type int and is popped from the 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 itableswitch 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 itableswitch instruction. Execution then continues at the target address.

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

Notes

If a virtual machine does not support the int data type, the itableswitch instruction will not be available.

## 7.5.67 iushr

Logical shift right int

Format

| *iushr* |
|---|

Forms

iushr = 82 (0x52)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

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 the result right by s bit positions, with zero extension, where s is the value of the low five 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, as if value2 were subjected to a bitwise logical AND with the mask value 0x1f.

If a virtual machine does not support the int data type, the iushr instruction will not be available.

## 7.5.68 ixor

Boolean XOR int

Format

| |
|---|
| *ixor* |

Forms

ixor = 88 (0x58)

Stack

…, value1.word1, value1.word2, value2.word1, value2.word2 ->
…, result.word1, result.word2

Description

Both value1 and value2 must be of type int. The values 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.

Notes

If a virtual machine does not support the int data type, the ixor instruction will not be available.

## 7.5.69 jsr

Jump subroutine

Format

| |
|---|
| *jsr* |
| *branchbyte1* |
| *branchbyte2* |

Forms

jsr = 113 (0x71)

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

The jsr instruction is used with the ret instruction in the implementation of the finally clause of the Java language. Note that jsr pushes the address onto the stack and ret gets it out of a local variable. This asymmetry is intentional.

## 7.5.70 new

Create new object

Format

| |
|---|
| *new* |
| *indexbyte1* |
| *indexbyte2* |

Forms

new = 143 (0x8f)

Stack

… ->
…, objectref

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, “Frames” on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The item at that index in the constant pool must be of type CONSTANT_Classref (Section 6.7.1, “CONSTANT_Classref” on page 6-16), a reference to a class or interface type. The reference is resolved and must result in a class type (it must not result in an interface type). Memory for a new instance of that class is allocated from the heap, and the instance variables of the new object are initialized to their default initial values. The objectref, a reference to the instance, is pushed onto the operand stack.

Notes

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

## 7.5.71 newarray

Create new array

Format

| |
|---|
| *newarray* |
| *atype* |

Forms

newarray = 144 (0x90)

Stack

…, count ->
…, arrayref

Description

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

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

**TABLE 7-4** Array Values

| **Array Type** | *atype* |
|---|---|
| `T_BOOLEAN` | `10` |
| `T_BYTE` | `11` |
| `T_SHORT` | `12` |
| `T_INT` | `13` |

A new array whose components are of type atype, of length count, is allocated from the heap. A reference arrayref to this new array object is pushed onto the operand stack. All of the elements of the new array are initialized to the default initial value for its type.

Runtime Exception

If count is less than zero, the newarray instruction throws a NegativeArraySizeException.

Notes

If a virtual machine does not support the int data type, the value of atype may not be 13 (array type = T_INT).

## 7.5.72 nop

Do nothing

Format

| |
|---|
| *nop* |

Forms

nop = 0 (0x0)

Stack

No change

Description

Do nothing.

## 7.5.73 pop

Pop top operand stack word

Format

| |
|---|
| *pop* |

Forms

pop = 59 (0x3b)

Stack

…, word ->
…

Description

The top word is popped from the operand stack. The pop instruction must not be used unless the word contains a 16-bit data type.

Notes

The pop instruction operates on an untyped word, ignoring the type of data it contains.

## 7.5.74 pop2

Pop top two operand stack words

Format

| |
|---|
| *pop2* |

Forms

pop2 = 60 (0x3c)

Stack

…, word2, word1 ->
…

Description

The top two words are popped from the operand stack.

The pop2 instruction must not be used unless each of word1 and word2 is a word that contains a 16-bit data type or both together are the two words of a single 32-bit datum.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the pop2 instruction operates on an untyped word, ignoring the type of data it contains.

## 7.5.75 putfield_<t>

Set field in object

Format

| *putfield_<t>* |
|---|
| *index* |

Forms

putfield_a = 135 (0x87)
putfield_b = 136 (0x88)
putfield_s = 137 (0x89)
putfield_i = 138 (0x8a)

Stack

…, objectref, value ->
…

OR

…, objectref, value.word1, value.word2 ->
…

Description

The unsigned index is used as an index into the constant pool of the current package (Section 3.5, “Frames” on page 3-3). The constant pool item at the index must be of type CONSTANT_InstanceFieldref (Section 6.7.2, “CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref” on page 6-18), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class. If the field is final, it must be declared in the current class.

The item must resolve to a field with a type that matches t, as follows:

- a field must be of type reference
- b field must be of type byte or type boolean
- s field must be of type short
- i field must be of type int

value must be of a type that is assignment compatible with the field descriptor (t) type.

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset[1]. The objectref, which must be of type reference, and the value are popped from the operand stack. If the field is of type byte or type boolean, the value is truncated to a byte. The field at the offset from the start of the object referenced by objectref is set to the value.

Runtime Exception

If objectref is null, the putfield_<t> instruction throws a NullPointerException.

Notes

In some circumstances, the putfield_<t> instruction may throw a SecurityException if the current context Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the putfield_i instruction will not be available.

## 7.5.76 putfield_<t>_this

Set field in current object

Format

| |
|---|
| *putfield_<t>_this* |
| *index* |

Forms

putfield_a_this = 181 (0xb5)
putfield_b_this = 182 (0xb6)
putfield_s_this = 183 (0xb7)
putfield_i_this = 184 (0xb8)

Stack

…, value ->
…

OR

1. The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.

…, value.word1, value.word2 ->
…

Description

The currently executing method must be an instance method that was invoked using the invokevirtual, invokeinterface or invokespecial instruction. The local variable at index 0 must contain a reference objectref to the currently executing method's this parameter. The unsigned index is used as an index into the constant pool of the current package (Section 3.5, "Frames" on page 3-3). The constant pool item at the index must be of type CONSTANT_InstanceFieldref (Section 6.7.2, "CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref" on page 6-18), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class. If the field is final, it must be declared in the current class.

The item must resolve to a field with a type that matches t, as follows:

- a field must be of type reference
- b field must be of type byte or type boolean
- s field must be of type short
- i field must be of type int

value must be of a type that is assignment compatible with the field descriptor (t) type.

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset[1]. The value is popped from the operand stack. If the field is of type byte or type boolean, the value is truncated to a byte. The field at the offset from the start of the object referenced by objectref is set to the value.

Runtime Exception

If objectref is null, the putfield_<t>_this instruction throws a NullPointerException.

Notes

1. The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.

In some circumstances, the putfield_<t>_this instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the putfield_i_this instruction will not be available.

## 7.5.77 putfield_<t>_w

Set field in object (wide index)

Format

| |
|---|
| *putfield<t>_w* |
| *indexbyte1* |
| *indexbyte2* |

Forms

putfield_a_w = 177 (0xb1)
putfield_b_w = 178 (0xb2)
putfield_s_w = 179 (0xb3)
putfield_i_w = 180 (0xb4)

Stack

…, objectref, value ->
…

OR

…, objectref, value.word1, value.word2 ->
…

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, “Frames” on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at the index must be of type CONSTANT_InstanceFieldref (Section 6.7.2, “CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref” on page 6-18), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class. If the field is final, it must be declared in the current class.

The item must resolve to a field with a type that matches t, as follows:

- a field must be of type reference
- b field must be of type byte or type boolean
- s field must be of type short
- i field must be of type int

value must be of a type that is assignment compatible with the field descriptor (t) type.

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset[1]. The objectref, which must be of type reference, and the value are popped from the operand stack. If the field is of type byte or type boolean, the value is truncated to a byte. The field at the offset from the start of the object referenced by objectref is set to the value.

Runtime Exception

If objectref is null, the putfield_<t>_w instruction throws a NullPointerException.

Notes

In some circumstances, the putfield_<t>_w instruction may throw a SecurityException if the current context (Section 3.4, "Contexts" on page 3-2) is not the owning context (Section 3.4, "Contexts" on page 3-2) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the putfield_i_w instruction will not be available.

## 7.5.78 putstatic_<t>

Set static field in class

1. The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.

Format

| |
|---|
| *putstatic_<t>* |
| *indexbyte1* |
| *indexbyte2* |

Forms

putstatic_a = 127 (0x7f)
putstatic_b = 128 (0x80)
putstatic_s = 129 (0x81)
putstatic_i = 130 (0x82)

Stack

…, value ->
…

OR

…, value.word1, value.word2 ->
…

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the constant pool of the current package (Section 3.5, "Frames" on page 3-3), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at the index must be of type CONSTANT_StaticFieldref (Section 6.7.3, "CONSTANT_StaticFieldref and CONSTANT_StaticMethodref" on page 6-19), a reference to a static field. If the field is final, it must be declared in the current class.

The item must resolve to a field with a type that matches t, as follows:

- a field must be of type reference
- b field must be of type byte or type boolean
- s field must be of type short
- i field must be of type int

value must be of a type that is assignment compatible with the field descriptor (t) type.

The width of a class field is determined by the field type specified in the instruction. The item is resolved, determining the class field. The value is popped from the operand stack. If the field is of type byte or type boolean, the value is truncated to a byte. The field is set to the value.

Notes

In some circumstances, the putstatic_a instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the object being stored in the field. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

If a virtual machine does not support the int data type, the putstatic_i instruction will not be available.

## 7.5.79 ret

Return from subroutine

Format

|  |
|---|
| *ret* |
| *index* |

Forms

ret = 114 (0x72)

Stack

No change

Description

The index is an unsigned byte that must be a valid index into the local variables of the current frame (Section 3.5, “Frames” on page 3-3). The local variable at index must contain a value of type returnAddress. The contents of the local variable are written into the Java Card virtual machine’s pc register, and execution continues there.

Notes

The ret instruction is used with the jsr instruction in the implementation of the finally keyword of the Java language. Note that jsr pushes the address onto the stack and ret gets it out of a local variable. This asymmetry is intentional.

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

## 7.5.80 return

Return void from method

Format

| *return* |
|---|

Forms

return = 122 (0x7a)

Stack

… ->
[empty]

Description

Any values on the operand stack of the current method are discarded. The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

## 7.5.81 s2b

Convert short to byte

Format

| *s2b* |
|---|

Forms

s2b = 91 (0x5b)

Stack

…, value ->
…, result

Description

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

Notes

The s2b instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of value. The result may also not have the same sign as value.

## 7.5.82 s2i

Convert short to int

Format

| *s2i* |
|---|

Forms

s2i = 92 (0x5c)

Stack

…, value ->
…, result.word1, result.word2

Description

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

Notes

The s2i instruction performs a widening primitive conversion. Because all values of type short are exactly representable by type int, the conversion is exact.

If a virtual machine does not support the int data type, the s2i instruction will not be available.

## 7.5.83 sadd

Add short

Format

| *sadd* |
|---|

Forms

sadd = 65 (0x41)

Stack

…, value1, value2 ->
…, result

Description

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

If a sadd instruction overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide two's-complement format. 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.

## 7.5.84 saload

Load short from array

Format

| |
|---|
| *saload* |

Forms

saload = 38 (0x46)

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 short. Both arrayref and index are popped from the operand stack. The short value in the component of the array at index is retrieved and pushed onto the top of the operand stack.

Runtime Exceptions

If arrayref is null, saload throws a NullPointerException.

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

Notes

In some circumstances, the saload instruction may throw a SecurityException if the current context (Section 3.4, “Contexts” on page 3-2) is not the owning context (Section 3.4, “Contexts” on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

## 7.5.85 sand

Boolean AND short

Format

| *sand* |
|---|

Forms

sand = 83 (0x53)

Stack

…, value1, value2 ->
…, result

Description

Both value1 and value2 are popped from the operand stack. A short result is calculated by taking the bitwise AND (conjunction) of value1 and value2. The result is pushed onto the operand stack.

## 7.5.86 sastore

Store into short array

Format

| *sastore* |
|---|

Forms

sastore = 57 (0x39)

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 and value must both be of type short. The arrayref, index and value are popped from the operand stack. The short value is stored as the component of the array indexed by index.

Runtime Exception

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.

Notes

In some circumstances, the sastore instruction may throw a SecurityException if the current context (Section 3.4, "Contexts" on page 3-2) is not the owning context (Section 3.4, "Contexts" on page 3-2) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, Version 2.2.2*.

## 7.5.87 sconst_<s>

Push short constant

Format

| |
|---|
| *sconst_<s>* |

Forms

sconst_m1 = 2 (0x2)
sconst_0 = 3 (0x3)
sconst_1 = 4 (0x4)
sconst_2 = 5 (0x5)
sconst_3 = 6 (0x6)
sconst_4= 7 (0x7)
sconst_5 = 8 (0x8)

Stack

… ->
…, <s>

Description

Push the short constant <s> (-1, 0, 1, 2, 3, 4, or 5) onto the operand stack.

## 7.5.88 sdiv

Divide short

Format

| |
|---|
| *sdiv* |

Forms

sdiv = 71 (0x47)

Stack

…, value1, value2 ->
…, result

Description

Both value1 and value2 must be of type short. The values are popped from the operand stack. The short result is the value of the Java expression value1 / value2. The result is pushed onto the operand stack.

A short division rounds towards 0; that is, the quotient produced for short values in n/d is a short value q whose magnitude is as large as possible while satisfying | d · q | <= | n |. Moreover, q is a positive when | n | >= | d | and n and d have the same sign, but q is negative when | n | >= | 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 the largest possible magnitude for the short type, and the divisor is –1, then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case.

Runtime Exception

If the value of the divisor in a short division is 0, sdiv throws an ArithmeticException.

## 7.5.89 sinc

Increment local short variable by constant

Format

| |
|---|
| *sinc* |
| *index* |
| *const* |

Forms

sinc = 89 (0x59)

Stack

No change

Description

The index is an unsigned byte that must be a valid index into the local variable of the current frame (Section 3.5, "Frames" on page 3-3). The const is an immediate signed byte. The local variable at index must contain a short. The value const is first sign-extended to a short, then the local variable at index is incremented by that amount.

## 7.5.90 sinc_w

Increment local short variable by constant

Format

| |
|---|
| *sinc_w* |
| *index* |
| *byte1* |
| *byte2* |

Forms

sinc_w = 150 (0x96)

Stack

No change

Description

The index is an unsigned byte that must be a valid index into the local variable of the current frame (Section 3.5, “Frames” on page 3-3). The immediate unsigned byte1 and byte2 values are assembled into a short const where the value of const is (byte1 << 8) | byte2. The local variable at index, which must contain a short, is incremented by const.

## 7.5.91 sipush

Push short

Format

| |
|---|
| *sipush* |
| *byte1* |
| *byte2* |

Forms

sipush = 19 (0x13)

Stack

… ->
…, value1.word1, value1.word2

Description

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

Notes

If a virtual machine does not support the int data type, the sipush instruction will not be available.

## 7.5.92 sload

Load short from local variable

Format

| |
|---|
| *sload* |
| *index* |

Forms

sload = 22 (0x16)

Stack

… ->
…, value

Description

The index is an unsigned byte that must be a valid index into the local variables of the current frame (Section 3.5, "Frames" on page 3-3). The local variable at index must contain a short. The value in the local variable at index is pushed onto the operand stack.

## 7.5.93 sload_<n>

Load short from local variable

Format

| |
|---|
| *sload_<n>* |

Forms

sload_0 = 28 (0x1c)
sload_1 = 29 (0x1d)
sload_2 = 30 (0x1e)
sload_3 = 31 (0x1f)

Stack

… ->
…, value

Description

The <n> must be a valid index into the local variables of the current frame (Section 3.5, "Frames" on page 3-3). The local variable at <n> must contain a short. The value in the local variable at <n> is pushed onto the operand stack.

Notes

Each of the sload_<n> instructions is the same as sload with an index of <n>, except that the operand <n> is implicit.

## 7.5.94 slookupswitch

Access jump table by key match and jump

Format

| |
|---|
| *slookupswitch* |
| *defaultbyte1* |
| *defaultbyte2* |
| *npairs1* |
| *npairs2* |
| *match-offset pairs…* |

Pair Format

| |
|---|
| *matchbyte1* |
| *matchbyte2* |
| *offsetbyte1* |
| *offsetbyte2* |

Forms

slookupswitch = 117 (0x75)

Stack

…, key ->
…

Description

A slookupswitch instruction is a variable-length instruction. Immediately after the slookupswitch opcode follow a signed 16-bit value default, an unsigned 16-bit value npairs, and then npairs pairs. Each pair consists of a short match and a signed 16-bit offset. Each of the signed 16-bit values is constructed from two unsigned bytes as (byte1 << 8) | byte2.

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

The key must be of type short and is popped from the operand stack and 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 slookupswitch 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 slookupswitch 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 slookupswitch instruction.

Notes

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

## 7.5.95 smul

Multiply short

Format

| |
|---|
| *smul* |

Forms

smul = 69 (0x45)

Stack

…, value1, value2 ->
…, result

Description

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

If a smul instruction overflows, then the result is the low-order bits of the mathematical product as a short. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two values.

## 7.5.96 sneg

Negate short

Format

| *sneg* |
|---|

Forms

sneg = 72 (0x4b)

Stack

…, value ->
…, result

Description

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

For short values, negation is the same as subtraction from zero. Because the Java Card 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 short results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

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

### 7.5.97 sor

Boolean OR short

Format

| *sor* |
|---|

Forms

sor = 85 (0x55)

Stack

…, value1, value2 ->
…, result

Description

Both value1 and value2 must be of type short. The values are popped from the operand stack. A short result is calculated by taking the bitwise inclusive OR of value1 and value2. The result is pushed onto the operand stack.

## 7.5.98 srem

Remainder short

Format

| |
|---|
| *srem* |

Forms

srem = 73 (0x49)

Stack

…, value1, value2 ->
…, result

Description

Both value1 and value2 must be of type short. The values are popped from the operand stack. The short result is the value of the Java expression 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 that the dividend is the negative short of largest possible magnitude for its type and the divisor is –1 (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

Runtime Exception

If the value of the divisor for a short remainder operator is 0, srem throws an ArithmeticException.

## 7.5.99 sreturn

Return short from method

Format

| *sreturn* |
|---|

Forms

sreturn = 120 (0x78)

Stack

…, value ->
[empty]

Description

The value must be of type short. It is popped from the operand stack of the current frame (Section 3.5, “Frames” on page 3-3) 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 virtual machine then reinstates the frame of the invoker and returns control to the invoker.

## 7.5.100 sshl

Shift left short

Format

| *sshl* |
|---|

Forms

sshl = 77 (0x4d)

Stack

…, value1, value2 ->
…, result

Description

Both value1 and value2 must be of type short. The values are popped from the operand stack. A short result is calculated by shifting value1 left by s bit positions, where s is the value of the low five 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.

The mask value of 0x1f allows shifting beyond the range of a 16-bit short value. It is used by this instruction, however, to ensure results equal to those generated by the Java instruction ishl.

## 7.5.101 sshr

Arithmetic shift right short

Format

*sshr*

Forms

sshr = 79 (0x4f)

Stack

…, value1, value2 ->
…, result

Description

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

Notes

The resulting value is Î(value1) / 2s°, where s is value2 & 0x1f. For nonnegative value1, this is equivalent (even if overflow occurs) to truncating short 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.

The mask value of 0x1f allows shifting beyond the range of a 16-bit short value. It is used by this instruction, however, to ensure results equal to those generated by the Java instruction ishr.

## 7.5.102 sspush

Push short

Format

| |
|---|
| *sspush* |
| *byte1* |
| *byte2* |

Forms

sspush = 17 (0x11)

Stack

… ->
…, value

Description

The immediate unsigned byte1 and byte2 values are assembled into a signed short where the value of the short is (byte1 << 8) | byte2. The resulting value is pushed onto the operand stack.

## 7.5.103 sstore

Store short into local variable

Format

| |
|---|
| *sstore* |
| *index* |

Forms

sstore = 41 (0x29)

Stack

…, value ->
…

Description

The index is an unsigned byte that must be a valid index into the local variables of the current frame (Section 3.5, “Frames” on page 3-3). The value on top of the operand stack must be of type short. It is popped from the operand stack, and the value of the local variable at index is set to value.

## 7.5.104 sstore_<n>

Store short into local variable

Format

| |
|---|
| *sstore_<n>* |

Forms

sstore_0 = 47 (0x2f)
sstore_1 = 48 (0x30)
sstore_2 = 49 (0x31)
sstore_3 = 50 (0x32)

Stack

…, value ->
…

Description

The <n> must be a valid index into the local variables of the current frame (Section 3.5, “Frames” on page 3-3). The value on top of the operand stack must be of type short. It is popped from the operand stack, and the value of the local variable at <n> is set to value.

## 7.5.105 ssub

Subtract short

Format

| |
|---|
| *ssub* |

Forms

ssub = 67 (0x43)

Stack

…, value1, value2 ->
…, result

Description

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

For short subtraction, a – b produces the same result as a + (–b). For short values, subtraction from zeros is the same as negation.

Despite the fact that overflow or underflow may occur, in which case the result may have a different sign than the true mathematical result, execution of a ssub instruction never throws a runtime exception.

## 7.5.106 stableswitch

Access jump table by short index and jump

Format

| |
|---|
| *stableswitch* |
| *defaultbyte1* |
| *defaultbyte2* |
| *lowbyte1* |
| *lowbyte2* |
| *highbyte1* |
| *highbyte2* |
| *jump offsets…* |

Offset Format

| |
|---|
| *offsetbyte1* |
| *offsetbyte2* |

Forms

stableswitch = 115 (0x73)

Stack

…, index ->
…

Description

A stableswitch instruction is a variable-length instruction. Immediately after the stableswitch opcode follow a signed 16-bit value default, a signed 16-bit value low, a signed 16-bit value high, and then high – low + 1 further signed 16-bit offsets. The value low must be less than or equal to high. The high – low + 1 signed 16-bit offsets are treated as a 0-based jump table. Each of the signed 16-bit values is constructed from two unsigned bytes as (byte1 << 8) | byte2.

The index must be of type short and is popped from the stack. If index is less than low or index is greater than high, than a target address is calculated by adding default to the address of the opcode of this stableswitch 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 stableswitch instruction. Execution then continues at the target address.

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

## 7.5.107 sushr

Logical shift right short

Format

| |
|---|
| *sushr* |

Forms

sushr = 81 (0x51)

Stack

…, value1, value2 ->
…, result

Description

Both value1 and value2 must be of type short. The values are popped from the operand stack. A short result is calculated by sign-extending value1 to 32 bits[1] and shifting the result right by s bit positions, with zero extension, where s is the value of the low five bits of value2. The resulting value is then truncated to a 16-bit result. 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, as if value2 were subjected to a bitwise logical AND with the mask value 0x1f.

The mask value of 0x1f allows shifting beyond the range of a 16-bit short value. It is used by this instruction, however, to ensure results equal to those generated by the Java instruction iushr.

## 7.5.108 swap_x

Swap top two operand stack words

Format

| |
|---|
| *swap_x* |
| *mn* |

Forms

swap_x = 64 (0x40)

Stack

…, wordM+N, …, wordM+1, wordM, …, word1 ->
…, wordM, …, word1, wordM+N, …, wordM+1

Description

The unsigned byte mn is used to construct two parameter values. The high nibble, (mn & 0xf0) >> 4, is used as the value m. The low nibble, (mn & 0xf), is used as the value n. Permissible values for both m and n are 1 and 2.

The top m words on the operand stack are swapped with the n words immediately below.

The swap_x instruction must not be used unless the ranges of words 1 through m and words m+1 through n each contain either a 16-bit data type, two 16-bit data types, a 32-bit data type, a 16-bit data type and a 32-bit data type (in either order), or two 32-bit data types.

1. Sign extension to 32 bits ensures that the result computed by this instruction will be exactly equal to that computed by the Java iushr instruction, regardless of the input values. In a Java Card virtual machine the expression "0xffff >>> 0x01" yields 0xffff, where ">>>" is performed by the `sushr` instruction. The same result is rendered by a Java virtual machine.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the swap_x instruction operates on untyped words, ignoring the types of data they contain.

If a virtual machine does not support the int data type, the only permissible value for both m and n is 1.

## 7.5.109 sxor

Boolean XOR short

Format

|  |
|---|
| *sxor* |

Forms

sxor = 87 (0x57)

Stack

…, value1, value2 ->
…, result

Description

Both value1 and value2 must be of type short. The values are popped from the operand stack. A short result is calculated by taking the bitwise exclusive OR of value1 and value2. The result is pushed onto the operand stack.

CHAPTER 8

# Tables of Instructions

The following pages contain lists of the APDU instructions recognized by the Java Card platform, organized by opcode value (TABLE 8-1) and by opcode mnemonic (TABLE 8-2).

**TABLE 8-1** Instructions by Opcode Value

| dec | hex | mnemonic | dec | hex | mnemonic |
|---|---|---|---|---|---|
| 0 | 00 | nop | 47 | 2F | sstore_0 |
| 1 | 01 | aconst_null | 48 | 30 | sstore_1 |
| 2 | 02 | sconst_m1 | 49 | 31 | sstore_2 |
| 3 | 03 | sconst_0 | 50 | 32 | sstore_3 |
| 4 | 04 | sconst_1 | 51 | 33 | istore_0 |
| 5 | 05 | sconst_2 | 52 | 34 | istore_1 |
| 6 | 06 | sconst_3 | 53 | 35 | istore_2 |
| 7 | 07 | sconst_4 | 54 | 36 | istore_3 |
| 8 | 08 | sconst_5 | 55 | 37 | aastore |
| 9 | 09 | iconst_m1 | 56 | 38 | bastore |
| 10 | 0A | iconst_0 | 57 | 39 | sastore |
| 11 | 0B | iconst_1 | 58 | 3A | iastore |
| 12 | 0C | iconst_2 | 59 | 3B | pop |
| 13 | 0D | iconst_3 | 60 | 3C | pop2 |
| 14 | 0E | iconst_4 | 61 | 3D | dup |
| 15 | 0F | iconst_5 | 62 | 3E | dup2 |
| 16 | 10 | bspush | 63 | 3F | dup_x |
| 17 | 11 | sspush | 64 | 40 | swap_x |

**TABLE 8-1** Instructions by Opcode Value *(Continued)*

| dec | hex | mnemonic | dec | hex | mnemonic |
|---|---|---|---|---|---|
| 18 | 12 | bipush | 65 | 41 | sadd |
| 19 | 13 | sipush | 66 | 42 | iadd |
| 20 | 14 | iipush | 67 | 43 | ssub |
| 21 | 15 | aload | 68 | 44 | isub |
| 22 | 16 | sload | 69 | 45 | smul |
| 23 | 17 | iload | 70 | 46 | imul |
| 24 | 18 | aload_0 | 71 | 47 | sdiv |
| 25 | 19 | aload_1 | 72 | 48 | idiv |
| 26 | 1A | aload_2 | 73 | 49 | srem |
| 27 | 1B | aload_3 | 74 | 4A | irem |
| 28 | 1C | sload_0 | 75 | 4B | sneg |
| 29 | 1D | sload_1 | 76 | 4C | ineg |
| 30 | 1E | sload_2 | 77 | 4D | sshl |
| 31 | 1F | sload_3 | 78 | 4E | ishl |
| 32 | 20 | iload_0 | 79 | 4F | sshr |
| 33 | 21 | iload_1 | 80 | 50 | ishr |
| 34 | 22 | iload_2 | 81 | 51 | sushr |
| 35 | 23 | iload_3 | 82 | 52 | iushr |
| 36 | 24 | aaload | 83 | 53 | sand |
| 37 | 25 | baload | 84 | 54 | iand |
| 38 | 26 | saload | 85 | 55 | sor |
| 39 | 27 | iaload | 86 | 56 | ior |
| 40 | 28 | astore | 87 | 57 | sxor |
| 41 | 29 | sstore | 88 | 58 | ixor |
| 42 | 2A | istore | 89 | 59 | sinc |
| 43 | 2B | astore_0 | 90 | 5A | iinc |
| 44 | 2C | astore_1 | 91 | 5B | s2b |
| 45 | 2D | astore_2 | 92 | 5C | s2i |
| 46 | 2E | astore_3 | 93 | 5D | i2b |
| 94 | 5E | i2s | 141 | 8D | invokestatic |

**TABLE 8-1** Instructions by Opcode Value *(Continued)*

| dec | hex | mnemonic | dec | hex | mnemonic |
|---|---|---|---|---|---|
| 95 | 5F | icmp | 142 | 8E | invokeinterface |
| 96 | 60 | ifeq | 143 | 8F | new |
| 97 | 61 | ifne | 144 | 90 | newarray |
| 98 | 62 | iflt | 145 | 91 | anewarray |
| 99 | 63 | ifge | 146 | 92 | arraylength |
| 100 | 64 | ifgt | 147 | 93 | athrow |
| 101 | 65 | ifle | 148 | 94 | checkcast |
| 102 | 66 | ifnull | 149 | 95 | instanceof |
| 103 | 67 | ifnonnull | 150 | 96 | sinc_w |
| 104 | 68 | if_acmpeq | 151 | 97 | iinc_w |
| 105 | 69 | if_acmpne | 152 | 98 | ifeq_w |
| 106 | 6A | if_scmpeq | 153 | 99 | ifne_w |
| 107 | 6B | if_scmpne | 154 | 9A | iflt_w |
| 108 | 6C | if_scmplt | 155 | 9B | ifge_w |
| 109 | 6D | if_scmpge | 156 | 9C | ifgt_w |
| 110 | 6E | if_scmpgt | 157 | 9D | ifle_w |
| 111 | 6F | if_scmple | 158 | 9E | ifnull_w |
| 112 | 70 | goto | 159 | 9F | ifnonnull_w |
| 113 | 71 | jsr | 160 | A0 | if_acmpeq_w |
| 114 | 72 | ret | 161 | A1 | if_acmpne_w |
| 115 | 73 | stableswitch | 162 | A2 | if_scmpeq_w |
| 116 | 74 | itableswitch | 163 | A3 | if_scmpne_w |
| 117 | 75 | slookupswitch | 164 | A4 | if_scmplt_w |
| 118 | 76 | ilookupswitch | 165 | A5 | if_scmpge_w |
| 119 | 77 | areturn | 166 | A6 | if_scmpgt_w |
| 120 | 78 | sreturn | 167 | A7 | if_scmple_w |
| 121 | 79 | ireturn | 168 | A8 | goto_w |
| 122 | 7A | return | 169 | A9 | getfield_a_w |
| 123 | 7B | getstatic_a | 170 | AA | getfield_b_w |
| 124 | 7C | getstatic_b | 171 | AB | getfield_s_w |

**TABLE 8-1** Instructions by Opcode Value *(Continued)*

| dec | hex | mnemonic | dec | hex | mnemonic |
|---|---|---|---|---|---|
| 125 | 7D | getstatic_s | 172 | AC | getfield_i_w |
| 126 | 7E | getstatic_i | 173 | AD | getfield_a_this |
| 127 | 7F | putstatic_a | 174 | AE | getfield_b_this |
| 128 | 80 | putstatic_b | 175 | AF | getfield_s_this |
| 129 | 81 | putstatic_s | 176 | B0 | getfield_i_this |
| 130 | 82 | putstatic_i | 177 | B1 | putfield_a_w |
| 131 | 83 | getfield_a | 178 | B2 | putfield_b_w |
| 132 | 84 | getfield_b | 179 | B3 | putfield_s_w |
| 133 | 85 | getfield_s | 180 | B4 | putfield_i_w |
| 134 | 86 | getfield_i | 181 | B5 | putfield_a_this |
| 135 | 87 | putfield_a | 182 | B6 | putfield_b_this |
| 136 | 88 | putfield_b | 183 | B7 | putfield_s_this |
| 137 | 89 | putfield_s | 184 | B8 | putfield_i_this |
| 138 | 8A | putfield_i | | | … |
| 139 | 8B | invokevirtual | 254 | FE | impdep1 |
| 140 | 8C | invokespecial | 255 | FF | impdep2 |

**TABLE 8-2** Instructions by Opcode Mnemonic

| mnemonic | dec | hex | mnemonic | dec | hex |
|---|---|---|---|---|---|
| aaload | 36 | 24 | iand | 84 | 54 |
| aastore | 55 | 37 | iastore | 58 | 3A |
| aconst_null | 1 | 01 | icmp | 95 | 5F |
| aload | 21 | 15 | iconst_0 | 10 | 0A |
| aload_0 | 24 | 18 | iconst_1 | 11 | 0B |
| aload_1 | 25 | 19 | iconst_2 | 12 | 0C |
| aload_2 | 26 | 1A | iconst_3 | 13 | 0D |
| aload_3 | 27 | 1B | iconst_4 | 14 | 0E |
| anewarray | 145 | 91 | iconst_5 | 15 | 0F |
| areturn | 119 | 77 | iconst_m1 | 9 | 09 |

**TABLE 8-2** Instructions by Opcode Mnemonic *(Continued)*

| mnemonic | dec | hex | mnemonic | dec | hex |
|---|---|---|---|---|---|
| arraylength | 146 | 92 | idiv | 72 | 48 |
| astore | 40 | 28 | if_acmpeq | 104 | 68 |
| astore_0 | 43 | 2B | if_acmpeq_w | 160 | A0 |
| astore_1 | 44 | 2C | if_acmpne | 105 | 69 |
| astore_2 | 45 | 2D | if_acmpne_w | 161 | A1 |
| astore_3 | 46 | 2E | if_scmpeq | 106 | 6A |
| athrow | 147 | 93 | if_scmpeq_w | 162 | A2 |
| baload | 37 | 25 | if_scmpge | 109 | 6D |
| bastore | 56 | 38 | if_scmpge_w | 165 | A5 |
| bipush | 18 | 12 | if_scmpgt | 110 | 6E |
| bspush | 16 | 10 | if_scmpgt_w | 166 | A6 |
| checkcast | 148 | 94 | if_scmple | 111 | 6F |
| dup | 61 | 3D | if_scmple_w | 167 | A7 |
| dup_x | 63 | 3F | if_scmplt | 108 | 6C |
| dup2 | 62 | 3E | if_scmplt_w | 164 | A4 |
| getfield_a | 131 | 83 | if_scmpne | 107 | 6B |
| getfield_a_this | 173 | AD | if_scmpne_w | 163 | A3 |
| getfield_a_w | 169 | A9 | ifeq | 96 | 60 |
| getfield_b | 132 | 84 | ifeq_w | 152 | 98 |
| getfield_b_this | 174 | AE | ifge | 99 | 63 |
| getfield_b_w | 170 | AA | ifge_w | 155 | 9B |
| getfield_i | 134 | 86 | ifgt | 100 | 64 |
| getfield_i_this | 176 | B0 | ifgt_w | 156 | 9C |
| getfield_i_w | 172 | AC | ifle | 101 | 65 |
| getfield_s | 133 | 85 | ifle_w | 157 | 9D |
| getfield_s_this | 175 | AF | iflt | 98 | 62 |
| getfield_s_w | 171 | AB | iflt_w | 154 | 9A |
| getstatic_a | 123 | 7B | ifne | 97 | 61 |
| getstatic_b | 124 | 7C | ifne_w | 153 | 99 |
| getstatic_i | 126 | 7E | ifnonnull | 103 | 67 |

**TABLE 8-2** Instructions by Opcode Mnemonic *(Continued)*

| mnemonic | dec | hex | mnemonic | dec | hex |
|---|---|---|---|---|---|
| getstatic_s | 125 | 7D | ifnonnull_w | 159 | 9F |
| goto | 112 | 70 | ifnull | 102 | 66 |
| goto_w | 168 | A8 | ifnull_w | 158 | 9E |
| i2b | 93 | 5D | iinc | 90 | 5A |
| i2s | 94 | 5E | iinc_w | 151 | 97 |
| iadd | 66 | 42 | iipush | 20 | 14 |
| iaload | 39 | 27 | iload | 23 | 17 |
| iload_0 | 32 | 20 | putstatic_s | 129 | 81 |
| iload_1 | 33 | 21 | ret | 114 | 72 |
| iload_2 | 34 | 22 | return | 122 | 7A |
| iload_3 | 35 | 23 | s2b | 91 | 5B |
| ilookupswitch | 118 | 76 | s2i | 92 | 5C |
| imul | 70 | 46 | sadd | 65 | 41 |
| ineg | 76 | 4C | saload | 38 | 26 |
| instanceof | 149 | 95 | sand | 83 | 53 |
| invokeinterface | 142 | 8E | sastore | 57 | 39 |
| invokespecial | 140 | 8C | sconst_0 | 3 | 03 |
| invokestatic | 141 | 8D | sconst_1 | 4 | 04 |
| invokevirtual | 139 | 8B | sconst_2 | 5 | 05 |
| ior | 86 | 56 | sconst_3 | 6 | 06 |
| irem | 74 | 4A | sconst_4 | 7 | 07 |
| ireturn | 121 | 79 | sconst_5 | 8 | 08 |
| ishl | 78 | 4E | sconst_m1 | 2 | 02 |
| ishr | 80 | 50 | sdiv | 71 | 47 |
| istore | 42 | 2A | sinc | 89 | 59 |
| istore_0 | 51 | 33 | sinc_w | 150 | 96 |
| istore_1 | 52 | 34 | sipush | 19 | 13 |
| istore_2 | 53 | 35 | sload | 22 | 16 |
| istore_3 | 54 | 36 | sload_0 | 28 | 1C |
| isub | 68 | 44 | sload_1 | 29 | 1D |

**TABLE 8-2** Instructions by Opcode Mnemonic *(Continued)*

| mnemonic | dec | hex | mnemonic | dec | hex |
|---|---|---|---|---|---|
| itableswitch | 116 | 74 | sload_2 | 30 | 1E |
| iushr | 82 | 52 | sload_3 | 31 | 1F |
| ixor | 88 | 58 | slookupswitch | 117 | 75 |
| jsr | 113 | 71 | smul | 69 | 45 |
| new | 143 | 8F | sneg | 75 | 4B |
| newarray | 144 | 90 | sor | 85 | 55 |
| nop | 0 | 00 | srem | 73 | 49 |
| pop | 59 | 3B | sreturn | 120 | 78 |
| pop2 | 60 | 3C | sshl | 77 | 4D |
| putfield_a | 135 | 87 | sshr | 79 | 4F |
| putfield_a_this | 181 | B5 | sspush | 17 | 11 |
| putfield_a_w | 177 | B1 | sstore | 41 | 29 |
| putfield_b | 136 | 88 | sstore_0 | 47 | 2F |
| putfield_b_this | 182 | B6 | sstore_1 | 48 | 30 |
| putfield_b_w | 178 | B2 | sstore_2 | 49 | 31 |
| putfield_i | 138 | 8A | sstore_3 | 50 | 32 |
| putfield_i_this | 184 | B8 | ssub | 67 | 43 |
| putfield_i_w | 180 | B4 | stableswitch | 115 | 73 |
| putfield_s | 137 | 89 | sushr | 81 | 51 |
| putfield_s_this | 183 | B7 | swap_x | 64 | 40 |
| putfield_s_w | 179 | B3 | sxor | 87 | 57 |
| putstatic_a | 127 | 7F | | | |
| putstatic_b | 128 | 80 | | | |
| putstatic_i | 130 | 82 | | | |

# Glossary

| | |
|---|---|
| **active applet instance** | an applet instance that is selected on at least one of the logical channels. |
| **AID (application identifier)** | defined by ISO 7816, a string used to uniquely identify card applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies. <br><br> A unique AID is assigned for each package. In addition, a unique AID is assigned for each applet in the package. The package AID and the default AID for each applet defined in the package are specified in the `CAP` file. They are supplied to the converter when the `CAP` file is generated. |
| **APDU** | an acronym for Application Protocol Data Unit as defined in ISO 7816-4. |
| **API** | an acronym for Application Programming Interface. The API defines calling conventions by which an application program accesses the operating system and other services. |
| **applet** | within the context of this document, a Java Card applet, which is the basic unit of selection, context, functionality, and security in Java Card technology. |
| **applet developer** | a person creating an applet using Java Card technology. |
| **applet execution context** | context of a package that contains currently active applet. |
| **applet firewall** | the mechanism that prevents unauthorized accesses to objects in contexts other than currently active context. |
| **applet package** | see *library package*. |
| **assigned logical channel** | the logical channel on which the applet instance is either the active applet instance or will become the active applet instance. |

| | |
|---|---|
| **atomic operation** | an operation that either completes in its entirety or no part of the operation completes at all. |
| **atomicity** | state in which a particular operation is atomic. Atomicity of data updates guarantee that data are not corrupted in case of power loss or card removal. |
| **ATR** | an acronym for Answer to Reset. An ATR is a string of bytes sent by the Java Card platform after a reset condition. |
| **basic logical channel** | logical channel 0, the only channel that is active at card reset. This channel is permanent and can never be closed. |
| **big-endian** | a technique of storing multibyte data where the high-order bytes come first. For example, given an 8-bit data item stored in big-endian order, the first bit read is considered the high bit. |
| **binary compatibility** | in a Java Card system, a change in a Java programming language package results in a new `CAP` file. A new `CAP` file is binary compatible with (equivalently, does not break compatibility with) a preexisting `CAP` file if another `CAP` file converted using the export file of the preexisting `CAP` file can link with the new `CAP` file without errors. |
| **bytecode** | machine-independent code generated by the compiler and executed by the Java virtual machine. |
| **CAD** | an acronym for Card Acceptance Device. The CAD is the device in which the card is inserted. |
| **CAP file** | the `CAP` file is produced by the Converter and is the standard file format for the binary compatibility of the Java Card platform. A `CAP` file contains an executable binary representation of the classes of a Java programming language package. The `CAP` file also contains the `CAP` file components (see also *CAP file component*). The `CAP` files produced by the converter are contained in Java™ Archive (JAR) files. |
| **CAP file component** | a Java Card platform `CAP` file consists of a set of components which represent a Java programming language package. Each component describes a set of elements in the Java programming language package, or an aspect of the `CAP` file. A complete `CAP` file must contain all of the required components: Header, Directory, Import, Constant Pool, Method, Static Field, and Reference Location<br><br>The following components are optional: the Applet, Export, and Debug. The Applet component is included only if one or more Applets are defined in the package. The Export component is included only if classes in other packages may import elements in the package defined. The Debug component is optional. It contains all of the data necessary for debugging a package. |
| **card session** | a card session begins with the insertion of the card into the CAD. The card is then able to exchange streams of APDUs with the CAD. The card session ends when the card is removed from the CAD. |
| **cast** | the explicit conversion from one data type to another. |

| | |
|---|---|
| **constant pool** | the constant pool contains variable-length structures representing various string constants, class names, field names, and other constants referred to within the `CAP` file and the Export File structure. Each of the constant pool entries, including entry zero, is a variable-length structure whose format is indicated by its first tag byte. There are no ordering constraints on entries in the constant pool entries. One constant pool is associated with each package.<br><br>There are differences between the Java platform constant pool and the Java Card technology-based constant pool. For example, in the Java platform constant pool there is one constant type for method references, while in the Java Card constant pool, there are three constant types for method references. The additional information provided by a constant type in Java Card technologies simplifies resolution of references. |
| **context** | protected object space associated with each applet package and Java Card RE. All objects owned by an applet belong to context of the applet's package. |
| **context switch** | a change from one currently active context to another. For example, a context switch is caused by an attempt to access an object that belongs to an applet instance that resides in a different package. The result of a context switch is a new currently active context. |
| **Converter** | a piece of software that preprocesses all of the Java programming language class files that make up a package, and converts the package to a `CAP` file. The Converter also produces an export file. |
| **currently active context** | when an object instance method is invoked, an owning context of this object becomes the currently active context. |
| **currently selected applet** | the Java Card RE keeps track of the currently selected Java Card applet. Upon receiving a SELECT FILE command with this applet’s AID, the Java Card RE makes this applet the currently selected applet. The Java Card RE sends all APDU commands to the currently selected applet. |
| **custom CAP file component** | a new component added to the `CAP` file. The new component must conform to the general component format. It is silently ignored by a Java Card virtual machine that does not recognize the component. The identifiers associated with the new component are recorded in the `custom_component` item of the `CAP` file's Directory component. |
| **default applet** | an applet that is selected by default on a logical channel when it is opened. If an applet is designated the default applet on a particular logical channel on the Java Card platform, it becomes the active applet by default when that logical channel is opened using the basic channel. |
| **EEPROM** | an acronym for Electrically Erasable, Programmable Read Only Memory. |
| **entry point objects** | see Java Card RE entry point objects. |

| | |
|---|---|
| **Export file** | a file produced by the Converter that represents the fields and methods of a package that can be imported by classes in other packages. |
| **externally visible** | in the Java Card platform, any classes, interfaces, their constructors, methods, and fields that can be accessed from another package according to the Java programming language semantics, as defined by the *Java Language Specification*, and Java Card API package access control restrictions (see *Java Language Specification*, section 2.2.1.1). |
| | Externally visible items may be represented in an export file. For a library package, all externally visible items are represented in an export file. For an applet package, only those externally visible items that are part of a shareable interface are represented in an export file. |
| **finalization** | the process by which a Java virtual machine (VM) allows an unreferenced object instance to release non-memory resources (for example, close and open files) prior to reclaiming the object's memory. Finalization is only performed on an object when that object is ready to be garbage collected (meaning, there are no references to the object). |
| | Finalization is not supported by the Java Card virtual machine. The method `finalize()` is not called automatically by the Java Card virtual machine. |
| **firewall** | see *applet firewall*. |
| **flash memory** | a type of persistent mutable memory. It is more efficient in space and power than EPROM. Flash memory can be read bit by bit but can be updated only as a block. Thus, flash memory is typically used for storing additional programs or large chunks of data that are updated as a whole. |
| **framework** | the set of classes that implement the API. This includes core and extension packages. Responsibilities include applet selection, sending APDU bytes, and managing atomicity. |
| **garbage collection** | the process by which dynamically allocated storage is automatically reclaimed during the execution of a program. |
| **heap** | a common pool of free memory usable by a program. A part of the computer's memory used for dynamic memory allocation, in which blocks of memory are used in an arbitrary order. The Java Card virtual machine's heap is not required to be garbage collected. Objects allocated from the heap are not necessarily reclaimed. |
| **installer** | the on-card mechanism to download and install `CAP` files. The installer receives executable binary from the off-card installation program, writes the binary into the smart card memory, links it with the other classes on the card, and creates and initializes any data structures used internally by the Java Card Runtime Environment. |
| **installation program** | the off-card mechanism that employs a card acceptance device (CAD) to transmit the executable binary in a `CAP` file to the installer running on the card. |

|  |  |
|---|---|
| **instance variables** | also known as non-static fields. |
| **instantiation** | in object-oriented programming, to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function. |
| **instruction** | a statement that indicates an operation for the computer to perform and any data to be used in performing the operation. An instruction can be in machine language or a programming language. |
| **internally visible** | items that are not externally visible. These items are not described in a package's export file, but some such items use private tokens to represent internal references. See also *externally visible*. |
| **JAR file** | an acronym for Java Archive file, which is a file format used for aggregating many files into one. |
| **Java Card Platform Remote Method Invocation** | a subset of the Java Platform Remote Method Invocation (RMI) system. It provides a mechanism for a client application running on the CAD platform to invoke a method on a remote object on the card. |
| **Java Card Runtime Environment (Java Card RE)** | consists of the Java Card virtual machine, the framework, and the associated native methods. |
| **Java Card Virtual Machine (Java Card VM)** | a subset of the Java virtual machine, which is designed to be run on smart cards and other resource-constrained devices. The Java Card VM acts an engine that loads Java class files and executes them with a particular set of semantics. |
| **Java Card RE entry point objects** | objects owned by the Java Card RE context that contain entry point methods. These methods can be invoked from any context and allow non-privileged users (applets) to request privileged Java Card RE system services. Java Card RE entry point objects can be either temporary or permanent:<br><br>**temporary -** references to temporary Java Card RE entry point objects cannot be stored in class variables, instance variables or array components. The Java Card RE detects and restricts attempts to store references to these objects as part of the firewall functionality to prevent unauthorized reuse. Examples of these objects are APDU objects and all Java Card RE-owned exception objects.<br><br>**permanent -** references to permanent Java Card RE entry point objects can be stored and freely reused. Examples of these objects are Java Card RE-owned AID instances. |

| | |
|---|---|
| **JDK™ software** | an acronym for Java Development Kit. The JDK software is a Sun Microsystems, Inc. product that provides the environment required for software development in the Java programming language. The JDK software is available for a variety of operating systems, for example Sun Microsystems Solaris™ OS and Microsoft Windows. |
| **library package** | a Java programming language package that does not contain any non-abstract classes that extend the class `javacard.framework.Applet`. An applet package contains one or more non-abstract classes that extend the `javacard.framework.Applet` class. |
| **local variable** | a data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method is a local variable and cannot be used outside the method. |
| **logical channel** | as seen at the card edge, works as a logical link to an application on the card. A logical channel establishes a communications session between a card applet and the terminal. Commands issued on a specific logical channel are forwarded to the active applet on that logical channel. For more information, see the *ISO/IEC 7816 Specification, Part 4*. (`http://www.iso.org`). |
| **MAC** | an acronym for Message Authentication Code. MAC is an encryption of data for security purposes. |
| **mask production (masking)** | refers to embedding the Java Card virtual machine, runtime environment, and applets in the read-only memory of a smart card during manufacture. |
| **method** | a procedure or routine associated with one or more classes in object-oriented languages. |
| **multiselectable applets** | implements the `javacard.framework.MultiSelectable` interface. Multiselectable applets can be selected on multiple logical channels at the same time. They can also accept other applets belonging to the same package being selected simultaneously. |
| **multiselected applet** | an applet instance that is selected and, therefore, active on more than one logical channel simultaneously. |
| **namespace** | a set of names in which all names are unique. |
| **native method** | a method that is not implemented in the Java programming language, but in another language. The `CAP` file format does not support native methods. |
| **nibble** | four bits. |
| **object-oriented** | a programming methodology based on the concept of an *object*, which is a data structure encapsulated with a set of routines, called *methods*, which operate on the data. |

**object owner** the applet instance within the currently active context when the object is instantiated. An object can be owned by an applet instance, or by the Java Card RE.

**objects** in object-oriented programming, unique instances of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.

**origin logical channel** the logical channel on which an APDU command is issued.

**owning context** the context in which an object is instantiated or created.

**package** a namespace within the Java programming language that can have classes and interfaces.

**PCD** an acronym for Proximity Coupling Device. The PCD is a contactless card reader device.

**persistent object** persistent objects and their values persist from one CAD session to the next, indefinitely. Objects are persistent by default. Persistent object values are updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized and deserialized, just that the objects are not lost when the card loses power.

**PIX** see *AID*.

**RAM (random access memory)** temporary working space for storing and modifying data. RAM is non-persistent memory; that is, the information content is not preserved when power is removed from the memory cell. RAM can be accessed an unlimited number of times and none of the restrictions of EEPROM apply.

**reference implementation** a fully functional and compatible implementation of a given technology. It enables developers to build prototypes of applications based on the technology.

**remote interface** an interface which extends, directly or indirectly, the interface `java.rmi.Remote`.

Each method declaration in the remote interface or its super-interfaces includes the exception `java.rmi.RemoteException` (or one of its superclasses) in its `throws` clause.

In a remote method declaration, if a remote object is declared as a return type, it is declared as the remote interface, not the implementation class of that interface.

In addition, Java Card RMI imposes additional constraints on the definition of remote methods. These constraints are as a result of the Java Card platform language subset and other feature limitations.

| | |
|---|---|
| **remote methods** | the methods of a remote interface. |
| **remote object** | an object whose remote methods can be invoked remotely from the CAD client. A remote object is described by one or more remote interfaces. |
| **RFU** | acronym for Reserved for Future Use. |
| **RID** | see *AID*. |
| **RMI** | an acronym for Remote Method Invocation. RMI is a mechanism for invoking instance methods on objects located on remote virtual machines (meaning, a virtual machine other than that of the invoker). |
| **ROM (read-only memory)** | memory used for storing the fixed program of the card. A smart card's ROM contains operating system routines as well as permanent data and user applications. No power is needed to hold data in this kind of memory. ROM cannot be written to after the card is manufactured. Writing a binary image to the ROM is called masking and occurs during the chip manufacturing process. |
| **runtime environment** | see *Java Card Runtime Environment (Java Card RE)*. |
| **shareable interface** | an interface that defines a set of shared methods. These interface methods can be invoked from an applet in one context when the object implementing them is owned by an applet in another context. |
| **shareable interface object (SIO)** | an object that implements the shareable interface. |
| **smart card** | a card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Unlike magnetic stripe cards, smart cards carry both processing power and information. They do not require access to remote databases at the time of a transaction. |
| **terminal** | a Card Acceptance Device that is typically a computer in its own right and can integrate a card reader as one of its components. In addition to being a smart card reader, a terminal can process data exchanged between itself and the smart card. |
| **thread** | the basic unit of program execution. A process can have several threads running concurrently each performing a different job, such as waiting for events or performing a time consuming job that the program doesn't need to complete before going on. When a thread has finished its job, it is suspended or destroyed.<br><br>The Java Card virtual machine can support only a single thread of execution. Java Card technology programs cannot use class `Thread` or any of the thread-related keywords in the Java programming language. |
| **transaction** | an atomic operation in which the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction. |

| | |
|---|---|
| **transient object** | the state of transient objects do not persist from one CAD session to the next, and are reset to a default state at specified intervals. Updates to the values of transient objects are not atomic and are not affected by transactions. |
| **verification** | a process performed on a `CAP` file that ensures that the binary representation of the package is structurally correct. |
| **word** | an abstract storage unit. A word is large enough to hold a value of type `byte`, `short`, `reference` or `returnAddress`. Two words are large enough to hold a value of `integer` type. |