Space Details

Key:	AVM2
Name:	ActionScript Virtual Machine 2 (AVM2)
Description:	
Creator (Creation Date):	ev94114 (Apr 15, 2010)
Last Modifier (Mod. Date):	ev94114 (Apr 15, 2010)

Available Pages

- bitxor
- call
- callmethod
- callproperty
- callproplex
- callpropvoid
- callstatic
- callsuper
- callsupervoid
- checkfilter
- coerce
- coerce_a
- coerce_s
- construct
- constructprop
- constructsuper
- convert_b
- convert_d
- convert_i
- convert_o
- convert_s
- convert_u
- debug
- debugfile
- debugline
- declocal
- declocal_i
- decrement
- decrement_i
- deleteproperty
- divide
- dup
- dxns
- dxnslate
- equals
- esc_xattr
- esc_xelem
- findproperty
- findpropstrict
- getdescendants
- getglobalscope
- getglobalslot

- getlex
- getlocal
- getlocal_n
- getproperty
- getscopeobject
- getslot
- getsuper
- greaterequals
- greaterthan
- hasnext
- hasnext2
- ifeq
- iffalse
- ifge
- ifgt
- ifle
- iflt
- ifne
- ifnge
- ifngt
- ifnle
- ifnlt
- ifstricteq
- ifstrictne
- iftrue
- in
- inclocal
- inclocal_i
- increment
- increment_i
- initproperty
- instanceof
- istype
- istypelate
- jump
- kill
- label
- lessequals
- lessthan
- lf32
- lf64
- li16

- li32
- li8
- lookupswitch
- lshift
- modulo
- multiply
- multiply_i
- negate
- negate_i
- newactivation
- newarray
- newcatch
- newclass
- newfunction
- newobject
- nextname
- nextvalue
- nop
- not
- pop
- popscope
- pushbyte
- pushdouble
- pushfalse
- pushint
- pushnamespace
- pushnan
- pushnull
- pushscope
- pushshort
- pushstring
- pushtrue
- pushuint
- pushundefined
- pushwith
- returnvalue
- returnvoid
- rshift
- setglobalslot
- setlocal
- setlocal_n
- setproperty

        - setslot
        - setsuper
        - sf32
        - sf64
        - si16
        - si32
        - si8
        - strictequals
        - subtract
        - subtract_i
        - swap
        - sxi_1
        - sxi_16
        - sxi_8
        - throw
        - typeof
        - urshift
    - 6. Hints for compiler writers
- Resources
- empty
- empty2
- empty3
- empty4

ActionScript Virtual Machine 2

This page last changed on Apr 15, 2010 by ev94114.

This is the home page for the ActionScript Virtual Machine 2 (AVM2) space.

1. Introduction

This page last changed on May 11, 2010 by john_rau.

The Adobe® ActionScript Virtual Machine 2, or AVM2 for short, was designed to execute programs written in the ActionScript 3.0 language. ActionScript 3.0 is based on ECMAScript, the international standardized programming language for scripting. ActionScript 3.0 is compliant with the ECMAScript Language Specification, Third Edition (ECMA-262). It also contains functionality based on ongoing work on ECMAScript Edition 4, occurring within the Ecma International standards body.

This document describes the operation of the AVM2 and defines the file formats, data structures, and instruction formats used by the AVM2.

1.1 Concepts

This page last changed on May 11, 2010 by john_rau.

The AVM2 was designed to support the ActionScript (AS) 3.0 language, and for the remaining chapters it is assumed that the reader is aware of the terminology and concepts of the language.

The following vocabulary and associated definitions are taken from the ActionScript 3.0 Language Specification and are presented only as a review of the material. For full details, refer to the language specification.

- Virtual Machine - A virtual machine is a mechanism that takes as its input the description of a computation and that performs that computation. For the AVM2, the input is in the form of an ABC file, which contains compiled programs; these comprise constant data, instructions from the AVM2 instruction set, and various kinds of metadata.
- Script - A script set of traits and an initializer method; a script populates a top-level environment with definitions and data.
- Bytecode, code - Bytecode or code is a specification of computation in the form of a sequence of simple actions on the virtual machine state.
- Scope - Scope is a mapping from names to locations, where no two names are the same. Scopes can nest, and nested scopes can contain bindings (associations between names and locations) that shadow the bindings of the nesting scope.
- Object - An object is an unordered collection of named properties, which are containers that hold values. A value in ActionScript 3.0 is either an Object reference or one of the special values `null` or `undefined`.
- Namespace - Namespaces are used to control the visibility of a set of properties independent of the major structure of the program.
- Class - A class is a named description of a group of objects. Objects are created from classes by instantiation.
- Inheritance - New classes can be derived from older classes by the mechanism known as inheritance or subclassing. The new class is called the derived class or subclass of the old class, and the old class is called the base class or superclass.
- Trait - A trait is a fixed-name property shared by all objects that are instances of the same class; a set of traits expresses the type of an object.
- Method - The word method is used with two separate meanings. One meaning is a method body, which is an object that contains code as well as data that belong to that code or that describe the code. The other meaning is a method closure, which is a method body together with a reference to the environment in which the closure was created. In this document, functions, constructors, ActionScript 3.0 class methods, and other objects that can be invoked are collectively referred to as method closures.
- Verification - The contents of an ABC file undergo verification when the file is loaded into the AVM2. The ABC file is rejected by the verifier if it does not conform to the AVM2 Overview. Verification is described in 3. Loading, linking, verification, and execution.
- Just-in-Time (JIT) Compiler - AVM2 implementations may contain an optional run-time compiler for transforming AVM2 instructions into processor-specific instructions. Although not an implementation requirement, employing a JIT compiler provides a performance benefit for many applications.

2. The structure of the ActionScript Virtual Machine

This page last changed on May 11, 2010 by john_rau.

2.1 Constant values

This page last changed on May 11, 2010 by john_rau.

The constant values of the AVM2 are one of the following types: int, uint, double, string, namespace, undefined, or null. The values of these types appear directly in the ABC file or in the instruction encodings. Important characteristics of these types are:

- int - This type is used to represent an integer valued number whose values are 32-bit signed two's complement integers. The range of values is from -2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31}-1$), inclusive.
- uint - This type is used for integer valued numbers with values that are 32-bit unsigned two's complement integers. The range of values is from 0 to 4,294,967,296 (2^{32}), inclusive.
- double - This type is used for capturing floating point numbers using 64-bit double precision IEEE 754 values as specified in IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard. 754-1985).
- string - This type represents a sequence of Unicode characters. Strings are represented in UTF-8 and can be as long as $2^{30}-1$ bytes.
- namespace - Namespaces tie a URI (represented internally by a string) to a trait. The relationship is unidirectional, meaning the namespace data type only contains a URI. Each namespace is also of a particular kind, and there are restrictions regarding the relationships between the trait and kind. These rules are defined later in this chapter.
- null - A singleton value representing "no object".
- undefined - A singleton value representing "no meaningful value". This constant value is allowed only in certain contexts.

The AVM2 utilizes several representations for values in its instruction encoding and in the ABC file in order to provide as compact an encoding as is required.

2.2 Virtual machine overview

This page last changed on Jun 21, 2010 by john_rau.

Computation in the AVM2 is based on executing the code of a method body in the context of method information, a local data area, a constant pool, a heap for non-primitive data objects created at run-time, and a run-time environment. Many data elements are static and are read at startup from an ABC file, whose structure is defined in 4. The ActionScript Byte Code (abc) format.

- The code for a method body is composed of instructions, defined in 5. AVM2 instructions. Each instruction modifies the state of the machine in some way, or has an effect on the external environment by means of input or output. [1]
- The method information determines how the method is used - for example, how default argument values should be substituted for missing arguments when the method is called.
- The local data area for the method consists of the operand stack, the scope stack, and the local registers.
    - The operand stack holds operands for the instructions and receives their results. Arguments are pushed onto the stack top or popped off the stack top. The top element always has address 0; the one below it has address 1, and so on. Stack addresses are not used except as a specification mechanism.
    - The scope stack is part of the run-time environment and holds objects that are to be searched by the AVM2 when an instruction is executed that calls for name lookup. Instructions push elements onto the scope stack as part of the implementation of exception handling, closure creation, and for the ActionScript 3.0 `with` statement.
    - The local registers hold parameter values, local variables in some cases, and temporaries.
- The constant pool holds constant values that are referenced, ultimately, by the instruction stream: numbers, strings, and various kinds of names.
- Instructions and the AVM2 can create new objects at run-time, and these objects are allocated in the heap. The only way to access the heap is through an object allocated in it. Objects in the heap that are no longer needed will eventually be reclaimed by the AVM2.
- The run-time environment logically consists of a chain of objects, and named properties on these objects are the locations found during a name lookup at run-time. Name lookup proceeds from the innermost (most recently pushed) scope toward the outermost (global) scope.
  The creation of a method closure causes the run-time environment that is current at the time of creation to be captured in the closure; when the closure is later invoked, that scope is made current, and will be extended by the code in the method body.

[1] In practice, the AVM2 may transform the code at run-time by means of a JIT, but this does not affect the semantics of execution, only its performance.

2.3 Names

This page last changed on Jun 21, 2010 by john_rau.

Names in the AVM are represented by a combination of an unqualified name and one or more namespaces. These are collectively called multinames. Multiname entries usually consist of a name index, and a namespace or namespace set index. Some multinames can have the name and/or namespace part resolved at runtime. There are a number of different types of multinames as described below. Properties of objects are always named by a simple QName (a pair of name and namespace). The other types of multinames are used to resolve properties at runtime.

RTQName, RTQNameL, and MultinameL are collectively referred to as runtime multinames.

QName (Qualified Name)

This is the simplest form of a multiname. It is a name with exactly one namespace, hence QName for qualified name. QName entries will have a name index followed by a namespace index. The name index is an index into the string constant pool, and the namespace index is an index into the namespace constant pool.

QNames are typically used to represent the names of variables, and for type annotations.

```
public var s : String;
```

This code will produce two QName entries, one for the variable s (public namespace, name "s") and one for the type String (public namespace, name "String").

RTQName (Runtime Qualified Name)

This is a runtime QName, where the namespace is not resolved until runtime. RTQName entries will have only a name index, which is an index into the string constant pool. The namespace is determined at runtime. When a RTQName is an operand to an opcode, there should be a namespace value on the stack the RTQName should use. So when the RTQName is used, the top value of the stack will be popped off, and the RTQName will use that as its namespace.

RTQNames are typically used for qualified names when the namespace is not known at compile time.

```
var ns = getANamespace();
x = ns::r;
```

This code will produce a RTQName entry for `ns::r`. It will have a name of "`r`" and code will be generated to push the value of `ns` onto the stack.

RTQNameL (Runtime Qualified Name Late)

This is a runtime QName, where both the name and namespace are resolved at runtime. When a RTQNameL is an operand to an opcode there will be a name and a namespace value on the stack. The name value on the stack must be of type String, and the namespace value on the stack must be of type Namespace.

RTQNameLs are typically used for qualified names when neither the name, nor the qualifier is known at compile time.

```
var x = getAName();
var ns = getANamespace();
```

```
w = ns::[x];
```

This code will produce a RTQNameL entry in the constant pool for `ns::x`. It has neither a name nor a namespace, but code will be generated to push the value of `ns` and `x` onto the stack.

Multiname (Multiple Namespace Name)

This is a multiname with a name and a namespace set. The namespace set is used to represent a collection of namespaces. Multiname entries will have a name index followed by a namespace set index. The name index is an index into the string constant pool, and the namespace set index is an index into the namespace set constant pool.

Multinames are typically used for unqualified names. In these cases all open namespaces are used for the multiname.

```
use namespace t;
trace(f);
```

This code will produce a multiname entry for `f`. It will have a name of "`f`" and a namespace set for all the open namespaces (the public namespace, the namespace `t`, and any private or internal namespaces open in that context). At runtime `f` could be resolved in any of the namespaces specified by the multiname.

MultinameL (Multiple Namespace Name Late)

This is a runtime multiname where the name is resolved at runtime. The namespace set is used to represent a collection of namespaces. MultinameL entries have a namespace set index. The namespace set index is an index into the namespace set constant pool. When a
MultinameL is an operand to an opcode there will be a name value on the stack. The name value on the stack must be of type String.

MultinameLs are typically used for unqualified names where the name is not known at compile time.

```
use namespace t;
trace(o[x]);
```

This code will produce a MultinameL entry. It will have no name, and will have a namespace set for all the open namespaces in that context. Code will be generated to push the value of `x` onto the stack, and that value will be used as the name.

Resolving multinames

Typically, the order of the search for resolving multinames is the object's declared traits, its dynamic properties, and finally the prototype chain. [2] The dynamic properties and prototype chain search will only happen if the multiname contains the public namespace (dynamic properties are always in the public namespace in ActionScript 3.0; a run-time error is signaled if an attempt is add a non-public property). If a search does not include one or more of these locations, it is noted in the text in the following chapters. Otherwise, you can assume that all three are searched in this order to resolve a multiname.

If the multiname is any type of QName, then the QName will resolve to the property with the same name and namespace as the QName. If no property has the same name and namespace as the QName, the QName is unresolved on that object.

If the multiname has a namespace set, then the object is searched for any properties whose name is the same as the multinames name, and whose namespace matches any of the namespaces in the multinames namespace set. Since the multiname may have more than one namespace, there could be multiple properties that match the multiname. If there are multiple properties that match a TypeError is raised since it is ambiguous which property the multiname is referring to. If no properties match, then the multiname is unresolved on that object.

[2] ECMAScript 3 supports prototyped-based inheritance. See ECMA 262 section 4.2.1 for a description of the prototype chain.

2.4 Method invocation notes

This page last changed on Jun 18, 2010 by john_rau.

When invoking a method in the AVM2, the first argument is always the "this" value to be used in the method. All methods take at least 1 argument (the "this" value), followed by any declared arguments.

When invoking the [[Call]] property, the behavior is different for different types of closures. A closure is an object that contains a reference to a method, and the [[Call]] property acts differently depending on whether it is a function, method, or class closure. A function closure is one that is of a global method that isn't associated with any instance of a class. A method closure contains an instance method of a class, and will always remember its original "this" value.

```
function f(){}
var a = f;  // a is a function closure

class C{
function m(){}
}
var q = new C();
var a = q.m;  // a is a method closure
```

If the closure is a function closure, then the first argument passed to [[Call]] is passed on to the method and gets used as the "this" value. If the first argument is `null` or `undefined`, then the global object will be used as the "this" value for the method.

If the closure is a method closure, then the first argument of [[Call]] will be ignored, and the saved "this" value for the method closure will be passed to the method as the first argument. A method closure records what its original "this" value was and always uses that instead of the first argument to [[Call]].

If the closure is a class closure, and there is 1 argument passed to [[Call]] (in addition to the "this" argument), then the call is treated as a type conversion, and the argument will be coerced to the type represented by the closure.

2.5 Instruction set summary

This page last changed on Jun 18, 2010 by john_rau.

This section provides an overview of the AVM2 instruction set. By convention, instructions specific to a specific data type are named with a suffix which indicates the data type on which it operates. Specifically, the following suffixes are used: `_b` (Boolean), `_a` (any), `_i` (int), `_d` (double), `_s` (string), `_u` (unsigned), and `_o` (object).

Load and store instructions

Local registers can be accessed using the following instructions: `getlocal, getlocal0, getlocal1, getlocal2, getlocal3, setlocal, setlocal0, setlocal1, setlocal2, setlocal3`.

Arithmetic instructions

The arithmetic instructions provide a full repertoire of mathematical operations. Zero, one, or more typically two operands are removed from the top of the stack and the result of the operation is pushed back onto the operand stack.

Addition is performed using one the following: `increment, increment_i, inclocal, inclocal_i, add, add_i`.

Subtraction is accomplished using the following: `decrement, decrement_i, declocal, declocal_i, subtract, subtract_i`.

Multiplication and division are achieved with `multiply, multiply_i, divide,` and `modulo`.

In order to reverse the sign of a value, the `negate` or `negate_i` instruction can be used.

There also exists a set of instructions that perform value comparisons on the top two entries on the stack replacing them with a `true` or `false` value. These include `equals, strictequals, lessthan, lessequals, greaterthan, greaterequals, istype, istypelate,` and `in`.

Bit manipulation instructions

Instructions that allow the bits of a value to be manipulated include `bitnot, bitand, bitor, bitxor, lshift, rshift, urshift`.

Prior to executing these instructions, the value to be operated upon is converted to an integer, if necessary.

Type conversion instructions

The ActionScript language is a loosely typed language where objects are freely converted into whatever types are necessary in order to complete an operation. In some cases explicit conversion is required and for those instances the coerce instructions are provided. These include `coerce, convert_b, coerce_a, convert_i, convert_d, coerce_s, convert_s, convert_u,` and `convert_o`.

Object creation and manipulation instructions

Entities are created by using one of the following instructions: `newclass, newobject, newarray, newactivation`.

In order to invoke an object's constructor the instructions `construct, constructsuper,` and `constructprop` are used.

Namespaces can be constructed dynamically using `dxns` and `dxnslate`.

Stack management instructions

A number of instructions provide direct access to manipulate values placed on the stack. Instructions that push a value directly include `pushnull, pushundefined, pushtrue, pushfalse, pushnan, pushbyte, pushshort, pushstring, pushint, pushdouble, pushscope,` and `pushnamespace`.

A value can be removed from the stack by using `pop`, while `dup` duplicates the value at the top of the stack, and similarly `swap` exchanges the top two values on the stack.

Control transfer instructions

The control transfer instructions transfer execution to an instruction other than the one immediately following the transfer instruction. The transfer can be unconditional or based upon a comparison operation that is implicit with the instruction.

The conditional branches instructions include `iflt, ifle, ifnlt, ifnle, ifgt, ifge, ifngt, ifnge, ifeq, ifne, ifstricteq, ifstrictne, iftrue,` and `iffalse`. These instructions perform any necessary type conversions in order to implement the compare; the conversion rules are outlined in ECMA-262.

The `label` instruction is used to mark the target position of backwards branch instruction. Thus the target location of every backwards branch instruction should land on a `label` instruction.

The `lookupswitch` instruction provides a compact form for encoding a multi-way compare expression.

Function invocation and return instructions

There are a number of instructions to invoke functions and methods. The call instruction implements a fully compliant rendition of `Function.prototype.call` of the ECMA-262 specification. To invoke object instance methods the `callmethod` instruction is utilized. Likewise for calling class, also known as static methods, `callstatic` exists. In order to invoke instance methods not on an object, but on its base class, `callsuper` is used. For named elements which are invoked as a method and for which the ActionScript compiler can validate the usage as such, `callproperty` and `callproplex` are available. The latter is for the case when the object for which the property being invoked exists on the stack.

For cases in which the return value of the call is never used, `callpropvoid` and `callsupervoid` can be used in place of `callproperty` and `callsuper`, respectively.

Exception instructions

An exception is thrown programmatically using the throw instruction. Exceptions can also be thrown by various AVM instructions when an abnormal condition is encountered.

The try/catch statement in the ActionScript language is translated into a table of intervals, and target instructions that are specified in the method body portion of the abc file. The table defines a range of instructions over which a given exception type may be caught. Thus if during the execution of a given set of instructions an exception is thrown and there is an associated entry in the exception table, program execution will continue at the target instruction specified in the table.

Debugging instructions

Unlike many traditional execution environments the debugging facilities of the AVM2 are tightly intertwined with a series of instructions that are placed directly in the execution stream. To track current file name and line number information `debugfile` and `debugline` are emitted at appropriate points in the instruction stream. In cases where additional debugging detail is required, the debug instruction is used. For example, the names of local variables are provided by this mechanism.

3. Loading, linking, verification, and execution

This page last changed on Jun 18, 2010 by john_rau.

3.1 Overview

This page last changed on Jun 18, 2010 by john_rau.

An ABC file is processed by the AVM2 in four logical phases, known as loading, linking, verification, and execution. These phases overlap and, in particular, verification overlaps with all of the other phases.

During loading, the ABC file is read into memory and decoded. For example, the constant pool is transformed to an in-memory data structure using different encodings. Verification at this stage is relative to the structure of the ABC file, which must conform to the definitions presented in the next chapter.

During linking, some names referenced from individual fields of the ABC file structure are resolved, and the resulting objects are linked together into a more complex data structure. For example, a class definition's "base class" field names the base class; resolving that name finds the definition of the base class, and a final class object for the derived class is created based on information from the two definition objects. Verification at this stage is relative to this web of objects: the names mentioned must resolve; the resulting traits sets must be coherent; and so on.

During execution, the bytecodes representing compiled code in the ABC file are run through an interpreter, thus performing computation. Verification at this stage is relative to the stream of instructions and the contents of the execution stack: instructions must not jump outside the bytecode array; instructions that require certain operand types can be applied only to operands whose known type is the correct one; the code must not use more stack and register space than it has reserved; and so on.

When verification fails during any of these phases, the AVM2 throws a VerifyError. VerifyErrors thrown during execution can be caught by the program.

The AVM2 interleaves linking with both loading and execution. When linking is performed during loading, forward references are precluded; for example, class definitions can only reference previously defined classes as base classes. In contrast, when linking is performed during execution, some references to undefined entities may not be flagged as errors; for example, a method that uses the name of a type that hasn't been defined may not cause a VerifyError to be thrown provided the method isn't invoked.

3.2 Loading and linking

This page last changed on Jun 18, 2010 by john_rau.

During the loading and linking phase, the following steps occur. (Note that a fair amount of linking is deferred until the execution phase.)

- Each ABC file is loaded into memory and decoded. This initial decoding verifies that the ABC file has the correct structure and that the fields that matter at this stage contain valid references to other parts of the ABC file.
- Trait objects are created for classes and scripts.
- Subclass/superclass relationships are resolved. The trait set of each class's superclass is merged into the trait set for the class, and interfaces in the class's interface set are looked up. Early resolution ensures that the inheritance graph is a tree.
- The constant pool is constructed. Each reference to another element of the constant pool is resolved (it must be in range for the correct type).
- Method bodies are linked with their method information (signature) structures.

3.3 Execution

This page last changed on Jun 18, 2010 by john_rau.

This section describes the state of the virtual machine as it is observed by the executing bytecode.

Program invocation and exit

One of the entries in the ABC file is an array of `script_info` entries (see the next chapter). Each of these entries contains a reference to an initialization method for the script and a set of traits to be defined in the script's environment. The last entry in that array is the entry point for the ABC file; that is, the last entry's initialization method contains the first bytecode that's run when the ABC file is executed.

The other scripts' initialization blocks are run on demand, when entities exported from those scripts are first referenced by the program during property lookup.

As far as the virtual machine is concerned, the initialization blocks are normal methods, and they should signal normal termination by executing one of the return instructions (`OP_returnvalue`, `OP_returnvoid`). The value returned by a script is ignored.

Class initialization

The class's static initializer will be run when the newclass instruction is executed on the class_info entry for the class.

Method entry

When a method body is first entered, its execution environment is set up in a particular way. In the following discussion, `method_info` and `method_body_info` refer to structures defined in the next chapter.

When control enters a method, three local data areas are allocated for it, as outlined in Chapter 2—an operand stack segment, a scope stack segment, and a set of local registers. The operand stack segment holds operands to the instructions; values are pushed onto that stack or popped off it by most instructions. The scope stack segments holds scope objects in which the virtual machine will look up names at execution time. Objects are pushed onto the scope stack by `OP_pushscope` and `OP_pushwith`, and popped off by `OP_popscope` and by the exception handling machinery. The local registers hold parameter values, local variables, and temporaries.

On method entry, the state of these data areas is as follows.

- The operand stack is empty and has room for `method_body_info.max_stack` values.
- The scope stack is empty and has room for `method_body_info.max_scope_stack` values.
- There are `method_body_info.local_count` registers.
- Register 0 holds the "this" object. This value is never `null`.
- Registers 1 through `method_info.param_count` holds parameter values coerced to the declared types of the parameters. If fewer than `method_body_info.local_count` values are supplied to the call then the remaining values are either the values provided by default value declarations (optional arguments) or the value `undefined`.
- If `NEED_REST` is set in `method_info.flags`, the `method_info.param_count+1` register is set up to reference an array that holds the superflous arguments.
- If `NEED_ARGUMENTS` is set in `method_info.flags`, the `method_info.param_count+1` register is set up to reference an "arguments" object that holds all the actual arguments: see ECMA-262 for more information. (The AVM2 is not strictly compatible with ECMA-262; it creates an `Array` object for the "arguments" object, whereas ECMA-262 requires a plain `Object`.)

Execution mechanics

Execution begins with the first instruction of the code in the `method_body_info`. The address of this first instruction is 0. The first byte of each instruction is the opcode, which is followed by zero or more bytes of operands. The instruction may modify the local data areas as well as objects on the heap, and it may create new objects on the heap either directly or indirectly. (The AVM2 instruction set is defined in 2.5 Instruction set summary)

Branch and jump instructions add a signed offset to the program counter to effectuate the branch. The base value for the program counter is the instruction address following the instruction.

Calling and returning

When one of the call instructions is executed, a new local data area is created for the called function. The called function has no access to the local data area of its caller. The actual parameter values are coerced to the types expected by the called function as part of the call protocol.

The return instructions transfer a single value from the returning function's stack, or the implied value `undefined`, to the caller. The returned value is coerced to the returning function's declared return type as part of the return protocol. The coerced value replaces the operands of the call instruction in the calling method's stack, and the called method's local data area is destroyed.

Exception handling

Exception handlers are defined by a table associated with each method. The table defines a range of bytecode addresses across which a particular handler is active, the bytecode address of the handler, and a type which is used to determine whether the handler will handle a particular exception.

When an exception is thrown, the call stack is unwound until a method is found that contains a handler which covers the current program counter and whose type is a supertype of the type of the object thrown. The method containing the handler is reactivated and the program counter is set to point to the first address of the handler. The value and scope stacks in the handling method are cleared before the handler is entered.

A `finally` clause is normally translated as an exception handler that accepts any type of value, which it catches and rethrows after the body of the `finally` block finishes executing.

3.4 Verification

This page last changed on Jun 18, 2010 by john_rau.

As noted earlier, verification does not happen all at once. Verification is often put off until a datum is actually needed or until some dependent object has been loaded, so that forward references will be possible.

The following list summarizes some of the verification errors signaled by the AVM2. (Not all verification errors are listed.)

- There must be no nonzero bits above bit 30 in a u30 value.
- No control flow instruction must cause control to be transferred outside the code vector or into the middle of another instruction.
- Multiple control flow instructions to the same instruction must have compatible scope stacks, operand stacks, and register values.
- Named types (for example, in coerce instructions or in a base class reference) must always be uniquely resolvable.
- Names may not reference index zero of the name pool unless explicitly specified for that particular name field.
- Some name fields (for example, `instance_info`) require the referenced name to be a QName.
- A class cannot subclass a final class or an interface.
- The interface set of a class cannot reference interface zero, and must reference interfaces (not classes).
- Method indices for `callmethod` and `callstatic` must provably be within the range of the receiver object's method table.
- When an instruction definition section contains wording along the lines of "<value> must be less than <constraint>", this usually implies a static constraint that is checked by the verifier.

4. The ActionScript Byte Code (abc) format

This page last changed on Jun 18, 2010 by john_rau.

Syntactically complete sections of ActionScript code are processed by a compiler into ActionScript Byte Code segments. These segments are described by the `abcFile` structure, which is defined below. The `abcFile` structure is the unit of loading and execution used by the AVM2.

The `abcFile` structure describes the interpretation of a block of 8-bit bytes. Despite the name, the contents of an `abcFile` does not need to be read from a file in the file system; it can be generated dynamically by a run-time compiler or other tools. The use of the word "file" is historical.

The `abcFile` structure comprises primitive data, structured data, and arrays of primitive and structured data. The following sections describe all the data formats.

Primitive data include integers and floating-point numbers encoded in various ways.

Structured data, including the `abcFile` itself, are presented here using a C-like structural notation, with individual named fields. Fields within this structure are in reality just sequences of bytes that are interpreted according to their type. The fields are stored sequentially without any padding or alignment.

4.1 Primitive data types

This page last changed on Jun 18, 2010 by john_rau.

Multi-byte primitive data are stored in little-endian order (less significant bytes precede more significant bytes). Negative integers are represented using two's complement.

- The type u8 represents a one-byte unsigned integer value.
- The type u16 represents a two-byte unsigned integer value.
- The type s24 represents a three-byte signed integer value.
- The type u30 represents a variable-length encoded 30-bit unsigned integer value.
- The types u32 and s32 represent variable-length encoded 32-bit unsigned and signed integer values respectively.
- The type d64 defines an 8-byte IEEE-754 floating point value. The high byte of the double value contains the sign and upper bits of the exponent, and the low byte contains the least significant bits of the significand.

The variable-length encoding for u30, u32, and s32 uses one to five bytes, depending on the magnitude of the value encoded. Each byte contributes its low seven bits to the value. If the high (eighth) bit of a byte is set, then the next byte of the abcFile is also part of the value. In the case of s32, sign extension is applied: the seventh bit of the last byte of the encoding is propagated to fill out the 32 bits of the decoded value.

4.10 Script

This page last changed on Jun 21, 2010 by john_rau.

The `script_info` entry is used to define characteristics of an ActionScript 3.0 script.

```
script_info
{
u30 init
u30 trait_count
traits_info trait[trait_count]
}
```

init

The `init` field is an index into the `method` array of the `abcFile`. It identifies a function that is to be invoked prior to any other code in this script.

trait_count, trait

The value of `trait_count` is the number of entries in the `trait` array. The `trait` array is the set of traits defined by the script.

4.11 Method body

This page last changed on Jun 21, 2010 by john_rau.

The `method_body_info` entry holds the AVM2 instructions that are associated with a particular method or function body. Some of the fields in this entry declare the maximum amount of resources the body will consume during execution. These declarations allow the AVM2 to anticipate the requirements of the method without analyzing the method body prior to execution. The declarations also serve as promises about the resource boundary within which the method has agreed to remain.
There can be fewer method bodies in the `method_body` table than than there are method signatures in the `method` table—some methods have no bodies. Therefore the `method_body` contains a reference to the method it belongs to, and other parts of the `abcFile` always reference the `method` table, not the `method_body` table.

```
method_body_info
{
u30 method
u30 max_stack
u30 local_count
u30 init_scope_depth
u30 max_scope_depth
u30 code_length
u8  code[code_length]
u30 exception_count
exception_info exception[exception_count]
u30 trait_count
traits_info trait[trait_count]
}
```

method

The `method` field is an index into the `method` array of the `abcFile`; it identifies the method signature with which this body is to be associated.

max_stack

The `max_stack` field is maximum number of evaluation stack slots used at any point during the execution of this body.

local_count

The `local_count` field is the index of the highest-numbered local register this method will use, plus one.

init_scope_depth

The `init_scope_depth` field defines the minimum scope depth, relative to max_scope_depth, that may be accessed within the method.

max_scope_depth

The `max_scope_depth` field defines the maximum scope depth that may be accessed within the method. The difference between `max_scope_depth` and `init_scope_depth` determines the size of the local scope stack.

code_length, code

The value of `code_length` is the number of bytes in the `code` array. The `code` array holds AVM2 instructions for this method body. The AVM2 instruction set is defined in 2.5 Instruction set summary.

exception_count, exception

The value of `exception_count` is the number of elements in the `exception` array. The `exception` array associates exception handlers with ranges of instructions within the code array (see below).

trait_count, trait

The value of `trait_count` is the number of elements in the `trait` array. The `trait` array contains all the traits for this method body (see above for more information on traits).

4.12 Exception

This page last changed on Jun 21, 2010 by john_rau.

The `exception_info` entry is used to define the range of ActionScript 3.0 instructions over which a particular exception handler is engaged.

```
exception_info
{
u30 from
u30 to
u30 target
u30 exc_type
u30 var_name
}
```

`from`

The starting position in the code field from which the exception is enabled.

`to`

The ending position in the code field after which the exception is disabled.

h2 `target`
The position in the code field to which control should jump if an exception of type `exc_type} is encountered while executing instructions that lie within the region {{from, to` of the `code` field.

`exc_type`

An index into the `string` array of the constant pool that identifies the name of the type of exception that is to be monitored during the reign of this handler. A value of zero means the any type ("*") and implies that this exception handler will catch any type of exception thrown.

`var_name`

This index into the `string` array of the constant pool defines the name of the variable that is to receive the exception object when the exception is thrown and control is transferred to `target` location. If the value is zero then there is no name associated with the exception object.

4.2 abcFile

This page last changed on Jun 21, 2010 by john_rau.

```
abcFile
{
u16 minor_version
u16 major_version
cpool_info constant_pool
u30 method_count
method_info method[method_count]
u30 metadata_count
metadata_info metadata[metadata_count]
u30 class_count
instance_info instance[class_count]
class_info class[class_count]
u30 script_count
script_info script[script_count]
u30 method_body_count
method_body_info method_body[method_body_count]
}
```

The `abcFile` structure describes an executable code block with all its constant data, type descriptors, code, and metadata. It comprises the following fields.

`minor_version, major_version`

The values of major_version and minor_version are the major and minor version numbers of the abcFile format. A change in the minor version number signifies a change in the file format that is backward compatible, in the sense that an implementation of the AVM2 can still make use of a file of an older version. A change in the major version number denotes an incompatible adjustment to the file format.

As of the publication of this overview, the major version is 46 and the minor version is 16.

`constant_pool`

The `constant_pool` is a variable length structure composed of integers, doubles, strings, namespaces, namespace sets, and multinames. These constants are referenced from other parts of the `abcFile` structure.

`method_count, method`

The value of `method_count` is the number of entries in the `method` array. Each entry in the `method` array is a variable length `method_info` structure. The array holds information about every method defined in this `abcFile`. The code for method bodies is held separately in the method_body array (see below). Some entries in `method` may have no body---this is the case for native methods, for example.

`metadata_count, metadata`

The value of `metadata_count` is the number of entries in the `metadata` array. Each `metadata` entry is a `metadata_info` structure that maps a name to a set of string values.

`class_count, instance, class`

The value of `class_count` is the number of entries in the `instance` and `class` arrays.

Each `instance` entry is a variable length `instance_info` structure which specifies the characteristics of object instances created by a particular class.

Each `class` entry defines the characteristics of a class. It is used in conjunction with the `instance` field to derive a full description of an AS Class.

script_count, script

The value of `script_count` is the number of entries in the `script` array. Each script entry is a `script_info` structure that defines the characteristics of a single script in this file. As explained in the previous chapter, the last entry in this array is the entry point for execution in the `abcFile`.

method_body_count, method_body

The value of `method_body_count` is the number of entries in the `method_body` array. Each `method_body` entry consists of a variable length `method_body_info` structure which contains the instructions for an individual method or function.

4.3 Constant pool

This page last changed on Jun 21, 2010 by john_rau.

The constant pool is a block of array-based entries that reflect the constants used by all methods. Each of the count entries (for example, `int_count`) must be one more than the number of entries in the corresponding array, and the first entry in the array is element "1". For all constant pools, the index "0" has a special meaning, typically a sensible default value. For example, the "0" entry is used to represent the empty sting (`""`), the any namespace, or the any type (`*`) depending on the context it is used in. When "0" has a special meaning it is described in the text below.

```
cpool_info
{
u30 int_count
s32 integer[int_count]
u30 uint_count
u32 uinteger[uint_count]
u30 double_count
d64 double[double_count]
u30 string_count
string_info string[string_count]
u30 namespace_count
namespace_info namespace[namespace_count]
u30 ns_set_count
ns_set_info ns_set[ns_set_count]
u30 multiname_count
multiname_info multiname[multiname_count]
}
```

If there is more than one entry in one of these arrays for the same entity, such as a name, the AVM may or may not consider those two entries to mean the same thing. The AVM currently guarantees that names flagged as belonging to the "private" namespace are treated as unique.

`int_count, integer`

The value of `int_count` is the number of entries in the `integer` array, plus one. The `integer` array holds integer constants referenced by the bytecode. The "0" entry of the `integer` array is not present in the `abcFile`; it represents the zero value for the purposes of providing values for optional parameters and field initialization.

`uint_count, uinteger`

The value of `uint_count` is the number of entries in the `uinteger` array, plus one. The `uinteger` array holds unsigned integer constants referenced by the bytecode. The "0" entry of the `uinteger` array is not present in the `abcFile`; it represents the zero value for the purposes of providing values for optional parameters and field initialization.

`double_count, double`

The value of `double_count` is the number of entries in the double array, plus one. The double array holds IEEE double-precision floating point constants referenced by the bytecode. The "0" entry of the double array is not present in the `abcFile`; it represents the NaN (Not-a-Number) value for the purposes of providing values for optional parameters and field initialization.

`string_count, string`

The value of `string_count` is the number of entries in the `string` array, plus one. The `string` array holds UTF-8 encoded strings referenced by the compiled code and by many other parts of the `abcFile`. In addition to describing string constants in programs, string data in the constant pool are used in the description of names of many kinds. Entry "0" of the `string` array is not present in the `abcFile`; it represents the empty string in most contexts but is also used to represent the "any" name in others (known as "`*`" in ActionScript).

namespace_count, namespace

The value of `namespace_count` is the number of entries in the `namespace` array, plus one. The `namespace` array describes the namespaces used by the bytecode and also for names of many kinds. Entry "0" of the `namespace` array is not present in the `abcFile`; it represents the "any" namespace (known as "*" in ActionScript).

ns_set_count, ns_set

The value of `ns_set_count` is the number of entries in the `ns_set` array, plus one. The `ns_set` array describes namespace sets used in the descriptions of multinames. The "0" entry of the `ns_set` array is not present in the `abcFile`.

multiname_count, multiname

The value of `multiname_count` is the number of entries in the `multiname` array, plus one. The `multiname` array describes names used by the bytecode. The "0" entry of the `multiname` array is not present in the `abcFile`.

Comments

One often encounters ABC formats with ns_set_count=0, e.g at http://www.adobe.com/devnet/flash/samples/time_1/index.html - but the way all other count fields in the constant pool are used, this should not happen, as the entry 0 is not present in the abcFile, so ns_set_count should always be at least 1.

Posted by fxrecuritylabs at Sep 03, 2010.

4.4 String

This page last changed on Jun 18, 2010 by john_rau.

A `string_info` element encodes a string of 16-bit characters on a length-and-data format. The meaning of each character is normally taken to be that of a Unicode 16-bit code point. The data are UTF 8 encoded. For more information on Unicode, see unicode.org.

```
string_info
{
u30 size
u8  utf8[size]
}
```


Namespace

A `namespace_info` entry defines a namespace. Namespaces have string names, represented by indices into the string array, and kinds. User-defined namespaces have kind `CONSTANT_Namespace` or `CONSTANT_ExplicitNamespace` and a non-empty name. System namespaces have empty names and one of the other kinds, and provides a means for the loader to map references to these namespaces onto internal entities.

```
namespace_info
{
u8  kind
u30 name
}
```

A single byte defines the type of entry that follows, thus identifying how the `name` field should be interpreted by the loader. The `name` field is an index into the string section of the constant pool. A value of zero denotes an empty string. The table below lists the legal values for `kind`.

Namespace Kind	Value
`CONSTANT_Namespace`	`0x08`
`CONSTANT_PackageNamespace`	`0x16`
`CONSTANT_PackageInternalNs`	`0x17`
`CONSTANT_ProtectedNamespace`	`0x18`
`CONSTANT_ExplicitNamespace`	`0x19`
`CONSTANT_StaticProtectedNs`	`0x1A`
`CONSTANT_PrivateNs`	`0x05`

Namespace set

An `ns_set_info` entry defines a set of namespaces, allowing the set to be used as a unit in the definition of multinames.

```
ns_set_info
{
u30 count
u30 ns[count]
```

```
}
```

The count field defines how many ns's are identified for the entry, while each ns is an integer that indexes into the `namespace` array of the constant pool. No entry in the ns array may be zero.

Multiname

A `multiname_info` entry is a variable length item that is used to define multiname entities used by the bytecode. There are many kinds of multinames. The kind field acts as a tag: its value determines how the loader should see the variable-length data field. The layout of the contents of the data field under a particular kind is described below by the `multiname_kind_` structures.

```
multiname_info
{
u8   kind
u8   data[]
}
```

Multiname Kind	Value
`CONSTANT_QName`	`0x07`
`CONSTANT_QNameA`	`0x0D`
`CONSTANT_RTQName`	`0x0F`
`CONSTANT_RTQNameA`	`0x10`
`CONSTANT_RTQNameL`	`0x11`
`CONSTANT_RTQNameLA`	`0x12`
`CONSTANT_Multiname`	`0x09`
`CONSTANT_MultinameA`	`0x0E`
`CONSTANT_MultinameL`	`0x1B`
`CONSTANT_MultinameLA`	`0x1C`

Those constants ending in "A" (such as `CONSTANT_QNameA`) represent the names of attributes.

QName

The `multiname_kind_QName` format is used for kinds `CONSTANT_QName` and CONSTANT_QNameA.

```
multiname_kind_QName
{
u30 ns
u30 name
}
```

The `ns` and `name` fields are indexes into the `namespace` and `string` arrays of the `constant_pool` entry, respectively. A value of zero for the ns field indicates the any ("*") namespace, and a value of zero for the `name` field indicates the any ("*") name.

RTQName

The `multiname_kind_RTQName` format is used for kinds `CONSTANT_RTQName` and `CONSTANT_RTQNameA`.

```
multiname_kind_RTQName
{
u30 name
}
```

The single field, `name`, is an index into the `string` array of the constant pool. A value of zero indicates the any ("*") name.

RTQNameL

The `multiname_kind_RTQNameL` format is used for kinds `CONSTANT_RTQNameL` and `CONSTANT_RTQNameLA`.

```
multiname_kind_RTQNameL
{
}
```

This kind has no associated data.

Multiname

The `multiname_kind_Multiname` format is used for kinds `CONSTANT_Multiname` and `CONSTANT_MultinameA`.

```
multiname_kind_Multiname
{
u30 name
u30 ns_set
}
```

The `name` field is an index into the `string` array, and the `ns_set` field is an index into the ns_set array. A value of zero for the `name` field indicates the any ("*") name. The value of `ns_set` cannot be zero.

MultinameL

The `multiname_kind_MultinameL` format is used for kinds `CONSTANT_MultinameL` and `CONSTANT_MultinameLA`.

```
multiname_kind_MultinameL
{
u30 ns_set
}
```

The `ns_set` field is an index into the `ns_set` array of the constant pool. The value of `ns_set` cannot be zero.

Comments

Other implementations, e.g. http://opensource.adobe.com/svn/opensource/flex/sdk/trunk/modules/swfutils/src/java/flash/swf/tools/AbcPrinter.java use three other Multiname kind identifiers:

- 0x13 - NameL
- 0x14 - NameL
- 0x1D - ???

These are not documented.

Posted by fxrecuritylabs at Sep 03, 2010.

4.5 Method signature

This page last changed on Jun 21, 2010 by john_rau.

The `method_info` entry defines the signature of a single method.

```
method_info
{
u30 param_count
u30 return_type
u30 param_type[param_count]
u30 name
u8  flags
option_info options
param_info param_names
}
```


Fields

The fields are as follows:

`param_count, param_type`

The `param_count` field is the number of formal parameters that the method supports; it also represents the length of the `param_type` array. Each entry in the `param_type` array is an index into the `multiname` array of the constant pool; the name at that entry provides the name of the type of the corresponding formal parameter. A zero value denotes the any ("*") type.

`return_type`

The `return_type` field is an index into the `multiname` array of the constant pool; the name at that entry provides the name of the return type of this method. A zero value denotes the any ("*") type.

`name`

The `name` field is an index into the `string` array of the constant pool; the string at that entry provides the name of this method. If the index is zero, this method has no name.

`flags`

The `flag` field is a bit vector that provides additional information about the method. The bits are described by the following table. (Bits not described in the table should all be set to zero.)

Name	Value	Meaning
`NEED_ARGUMENTS`	`0x01`	Suggests to the run-time that an "arguments" object (as specified by the ActionScript 3.0 Language Reference) be created. Must not be used together with `NEED_REST`. See 3. Loading, linking, verification, and execution.
`NEED_ACTIVATION`	`0x02`	

		Must be set if this method uses the `newactivation` opcode.
`NEED_REST`	`0x04`	This flag creates an ActionScript 3.0 rest arguments array. Must not be used with `NEED_ARGUMENTS`. See 3. Loading, linking, verification, and execution.
`HAS_OPTIONAL`	`0x08`	Must be set if this method has optional parameters and the options field is present in this `method_info` structure.
`SET_DXNS`	`0x40`	Must be set if this method uses the `dxns` or `dxnslate` opcodes.
`HAS_PARAM_NAMES`	`0x80`	Must be set when the `param_names` field is present in this `method_info` structure.

`options`

This entry may be present only if the `HAS_OPTIONAL` flag is set in `flags`.

`param_names`

This entry may be present only if the `HAS_PARAM_NAMES` flag is set in `flags`.

Optional parameters

The `option_info` entry is used to define the default values for the optional parameters of the method. The number of optional parameters is given by `option_count`, which must not be zero nor greater than the `parameter_count` field of the enclosing `method_info` structure.

```
option_info
{
u30 option_count
option_detail option[option_count]
}
```

```
option_detail
{
u30 val
u8  kind
}
```

Each optional value consists of a kind field that denotes the type of value represented, and a `val` field that is an index into one of the array entries of the constant pool. The correct array is selected based on the kind.

Constant Kind	Value	Entry
`CONSTANT_Int`	`0x03`	`integer`
`CONSTANT_UInt`	`0x04`	`uinteger`
`CONSTANT_Double`	`0x06`	`double`

CONSTANT_Utf8	0x01	string
CONSTANT_True	0x0B	-
CONSTANT_False	0x0A	-
CONSTANT_Null	0x0C	-
CONSTANT_Undefined	0x00	-
CONSTANT_Namespace	0x08	namespace
CONSTANT_PackageNamespace	0x16	namespace
CONSTANT_PackageInternalNs	0x17	Namespace
CONSTANT_ProtectedNamespace	0x18	Namespace
CONSTANT_ExplicitNamespace	0x19	Namespace
CONSTANT_StaticProtectedNs	0x1A	Namespace
CONSTANT_PrivateNs	0x05	namespace

Parameter names

The `param_names` entry is available only when the `HAS_PARAM_NAMES` bit is set in the flags. Each param_info element of the array is an index into the constant pool's `string` array. The parameter name entry exists solely for external tool use and is not used by the AVM2.

```
param_info
{
u30 param_name[param_count]
}
```

4.6 metadata_info

This page last changed on Jun 21, 2010 by john_rau.

The `metadata_info` entry provides a means of embedding arbitrary key /value pairs into the ABC file. The AVM2 will ignore all such entries.

```
metadata_info
{
u30 name
u30 item_count
item_info items[item_count]
}
```

The `name` field is an index into the `string` array of the constant pool; it provides a name for the metadata entry. The value of the name field must not be zero. Zero or more items may be associated with the entry; `item_count` denotes the number of items that follow in the `items` array.

```
item_info
{
u30 key
u30 value
}
```

The `item_info` entry consists of `item_count` elements that are interpreted as key/value pairs of indices into the string table of the constant pool. If the value of `key` is zero, this is a keyless entry and only carries a value.

4.7 Instance

This page last changed on Jun 21, 2010 by john_rau.

The `instance_info` entry is used to define the characteristics of a run-time object (a class instance) within the AVM2. The corresponding `class_info` entry is used in order to fully define an ActionScript 3.0 Class.

```
instance_info
{
u30 name
u30 super_name
u8  flags
u30 protectedNs
u30 intrf_count
u30 interface[intrf_count]
u30 iinit
u30 trait_count
traits_info trait[trait_count]
}
```

name

The `name` field is an index into the `multiname` array of the constant pool; it provides a name for the class. The entry specified must be a QName.

super_name

The `super_name` field is an index into the `multiname` array of the constant pool; it provides the name of the base class of this class, if any. A value of zero indicates that this class has no base class.

flags

The `flags` field is used to identify various options when interpreting the `instance_info` entry. It is bit vector; the following entries are defined. Other bits must be zero.

Name	Value	Meaning
`CONSTANT_ClassSealed`	`0x01`	The class is sealed: properties can not be dynamically added to instances of the class.
`CONSTANT_ClassFinal`	`0x02`	The class is final: it cannot be a base class for any other class.
`CONSTANT_ClassInterface`	`0x04`	The class is an interface.
`CONSTANT_ClassProtectedNs`	`0x08`	The class uses its protected namespace and the `protectedNs` field is present in the `interface_info` structure.

protectedNs

This field is present only if the `CONSTANT_ProtectedNs` bit of `flags` is set. It is an index into the `namespace` array of the constant pool and identifies the namespace that serves as the protected namespace for this class.

intrf_count, interface

The value of the `intrf_count` field is the number of entries in the `interface` array. The interface array contains indices into the `multiname` array of the constant pool; the referenced names specify the interfaces implemented by this class. None of the indices may be zero.

iinit

This is an index into the `method` array of the `abcFile`; it references the method that is invoked whenever an object of this class is constructed. This method is sometimes referred to as an instance initializer.

trait_count, trait

The value of `trait_count` is the number of elements in the `trait` array. The `trait` array defines the set of traits of a class instance. The next section defines the meaning of the `traits_info` structure.

4.8 Trait

This page last changed on Jun 21, 2010 by john_rau.

A trait is a fixed property of an object or class; it has a name, a type, and some associated data. The `traits_info` structure bundles these data.

```
traits_info
{
u30 name
u8  kind
u8  data[]
u30 metadata_count
u30 metadata[metadata_count]
}
```


Fields

`name`

The name field is an index into the `multiname` array of the constant pool; it provides a name for the trait. The value can not be zero, and the `multiname` entry specified must be a QName.

`kind`

The kind field contains two four-bit fields. The lower four bits determine the kind of this trait. The upper four bits comprise a bit vector providing attributes of the trait. See the following tables and sections for full descriptions.

`data`

The interpretation of the `data` field depends on the type of the trait, which is provided by the low four bits of the `kind` field. See below for a full description.

`metadata_count, metadata`

These fields are present only if `ATTR_Metadata` is present in the upper four bits of the `kind` field.

The value of the `metadata_count` field is the number of entries in the `metadata` array. That array contains indices into the `metadata` array of the `abcFile`.

Summary of trait types

The following table summarizes the trait types.

Type	Value
`Trait_Slot`	`0`

Trait_Method	1
Trait_Getter	2
Trait_Setter	3
Trait_Class	4
Trait_Function	5
Trait_Const	6

Slot and const traits

A kind value of `Trait_Slot` (0) or `Trait_Const` (6) requires that the data field be read using trait_slot, which takes the following form:

```
trait_slot
{
u30 slot_id
u30 type_name
u30 vindex
u8  vkind
}
```

slot_id

The `slot_id` field is an integer from 0 to N and is used to identify a position in which this trait resides. A value of 0 requests the AVM2 to assign a position.

type_name

This field is used to identify the type of the trait. It is an index into the `multiname` array of the `constant_pool`. A value of zero indicates that the type is the any type `( * )`.

vindex

This field is an index that is used in conjunction with the `vkind` field in order to define a value for the trait. If it is 0, `vkind` is empty; otherwise it references one of the tables in the constant pool, depending on the value of `vkind`.

vkind

This field exists only when `vindex` is non-zero. It is used to determine how `vindex` will be interpreted. See the "Constant Kind" table above for details.

Class traits

A kind value of Trait_Class (0x04) implies that the trait_class entry should be used.

```
trait_class
{
u30 slot_id
u30 classi
}
```

slot_id

The `slot_id` field is an integer from 0 to N and is used to identify a position in which this trait resides. A value of 0 requests the AVM2 to assign a position.

class

The `classi` field is an index that points into the `class` array of the `abcFile` entry.

Function traits

A kind value of `Trait_Function` (0x05) implies that the `trait_function` entry should be used.

```
trait_function
{
u30 slot_id
u30 function
}
```

slot_id

The `slot_id` field is an integer from 0 to N and is used to identify a position in which this trait resides. A value of 0 requests the AVM2 to assign a position.

function

The `function` field is an index that points into the `method` array of the `abcFile` entry.

Method, getter, and setter traits

A kind value of `Trait_Method` (0x01), `Trait_Getter` (0x02) or `Trait_Setter` (0x03) implies that the `trait_method` entry should be used.

```
trait_method
{
u30 disp_id
u30 method
}
```

disp_id

The `disp_id` field is a compiler assigned integer that is used by the AVM2 to optimize the resolution of virtual function calls. An overridden method must have the same `disp_id` as that of the method in the base class. A value of zero disables this optimization.

method

The `method` field is an index that points into the `method` array of the `abcFile` entry.

Trait attributes

As previously mentioned the upper nibble of the kind field is used to encode attributes. A description of how the attributes are interpreted for each `kind` is outlined below. Any other combination of `attribute` with `kind` is ignored.

Attributes	Value	
`ATTR_Final`	`0x1`	Is used with `Trait_Method`, `Trait_Getter` and `Trait_Setter`. It marks a method that cannot be overridden by a sub-class
`ATTR_Override`	`0x2`	Is used with `Trait_Method`, `Trait_Getter` and `Trait_Setter`. It marks a method that has been overridden in this class
`ATTR_Metadata`	`0x4`	Is used to signal that the fields `metadata_count` and `metadata` follow the `data` field in the `traits_info` entry

4.9 Class

This page last changed on Jun 21, 2010 by john_rau.

The `class_info` entry is used to define characteristics of an ActionScript 3.0 class.

```
class_info
{
u30 cinit
u30 trait_count
traits_info traits[trait_count]
}
```

cinit

This is an index into the `method` array of the `abcFile`; it references the method that is invoked when the class is first created. This method is also known as the static initializer for the class.

trait_count, trait

The value of `trait_count` is the number of entries in the `trait` array. The `trait` array holds the traits for the class (see above for information on traits).

5. AVM2 instructions

This page last changed on Jun 22, 2010 by john_rau.

The AVM2 instruction descriptions follow the following format.

Operation

Brief description of the instruction.

Format

A description of the instruction with any operands that appear with it in the code.

```
instruction
operand1
operand2
...
```

Forms

```
instruction = opcode
```

Stack

A description of the stack before and after the instruction is executed. The stack top is on the right; portions marked ... are not altered by the instruction.

```
..., value1, value2 => ..., value3
```

Description

A detailed description of the instruction, including information about the effect on the stack, information of the operands, result of the instruction, etc.

Runtime exceptions

A description of any errors that may be thrown by this instruction. A number of these instructions may invoke operations behind the scene. For example, name resolution can fail to find a name or it can resolve it ambiguously; value conversion can run arbitrary user code that may fail and therefore throw exceptions. To the program it will appear as if the instruction threw those exceptions, but they will not be noted in the description of the instruction.

Notes

Additional information that may be useful.

add

This page last changed on Jun 23, 2010 by john_rau.

Operation

Add two values.

Format

```
add
```

Forms

```
add = 160 (0xa0)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack and add them together as specified in ECMA-262 section 11.6 and as extended in ECMA-357 section 11.4. The algorithm is briefly described below.

1. If value1 and value2 are both Numbers, then set value3 to the result of adding the two number values. See ECMA-262 section 11.6.3 for a description of adding number values.
2. If value1 or value2 is a String or a Date, convert both values to String using the ToString algorithm described in ECMA-262 section 9.8. Concatenate the string value of value2 to the string value of value1 and set value3 to the new concatenated String.
3. If value1 and value2 are both of type XML or XMLList, construct a new XMLList object, then call /[/[Append]](value1), and then /[/[Append]](value2). Set value3 to the new XMLList. See ECMA-357 section 9.2.1.6 for a description of the /[/[Append]] method.
4. If none of the above apply, convert value1 and value2 to primitives. This is done by calling ToPrimitive with no hint. This results in value1_primitive and value2_primitive. If value1_primitive or value2_primitive is a String then convert both to Strings using the ToString algorithm (ECMA-262 section 9.8), concatenate the results, and set value3 to the concatenated String. Otherwise convert both to Numbers using the ToNumber algorithm (ECMA-262 section 9.3), add the results, and set value3 to the result of the addition.

Push value3 onto the stack.

Notes

For more information, see ECMA-262 section 11.6 ("Additive Operators") and ECMA-357 section 11.4.

add_i

This page last changed on Jun 23, 2010 by john_rau.

Operation

Add two integer values.

Format

```
add_i
```

Forms

```
add_i = 197 (0xc5)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack and convert them to int values using the ToInt32 algorithm (ECMA-262 section 9.5). Add the two int values and push the result onto the stack.

astype

This page last changed on Jun 23, 2010 by john_rau.

Operation

Return the same value, or null if not of the specified type.

Format

```
astype
index
```

Forms

```
astype = 134 (0x86)
```

Stack

..., value => ..., value

Description

`index` is a u30 that must be an index into the multiname constant pool. The multiname at index must not be a runtime multiname, and must be the name of a type.

Pop value off of the stack. If value is of the type specified by the multiname, push value back onto the stack. If value is not of the type specified by the multiname, then push null onto the stack.

astypelate

Operation

Return the same value, or null if not of the specified type.

Format

```
astypelate
```

Forms

```
astypelate = 135 (0x87)
```

Stack

..., value, class => ..., value

Description

Pop class and value off of the stack. class should be an object of type Class. If value is of the type specified by class, push value back onto the stack. If value is not of the type specified by class, then push null onto the stack.

Runtime exceptions

A TypeError is thrown if class is not of type Class.

bitand

Operation

Bitwise and.

Format

```
bitand
```

Forms

```
bitand = 168 (0xa8)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack. Convert value1 and value2 to integers, as per ECMA-262 section 11.10, and perform a bitwise and (&) on the two resulting integer values. Push the result onto the stack.

bitnot

This page last changed on Jun 23, 2010 by john_rau.

Operation

Bitwise not.

Format

```
bitnot
```

Forms

```
bitnot = 151 (0x97)
```

Stack

..., value => ..., ~value

Description

Pop value off of the stack. Convert value to an integer, as per ECMA-262 section 11.4.8, and then apply the bitwise complement operator (~) to the integer. Push the result onto the stack.

bitor

This page last changed on Jun 23, 2010 by john_rau.

Operation

Bitwise or.

Format

```
bitor
```

Forms

```
bitor = 169 (0xa9)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack. Convert value1 and value2 to integers, as per ECMA-262 section 11.10, and perform a bitwise or (|) on the two resulting integer values. Push the result onto the stack.

bitxor

This page last changed on Jun 23, 2010 by john_rau.

Operation

Bitwise exclusive or.

Format

```
bitxor
```

Forms

```
bitxor = 170 (0xaa)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack. Convert value1 and value2 to integers, as per ECMA-262 section 11.10, and perform a bitwise exclusive or (^) on the two resulting integer values. Push the result onto the stack.

call

This page last changed on Jun 23, 2010 by john_rau.

Operation

Call a closure.

Format

```
call
arg_count
```

Forms

```
call = 65 (0x41)
```

Stack

..., function, receiver, arg1, arg2, ..., argn => ..., value

Description

arg_count is a u30 that is the number of arguments present on the stack for the call. function is the closure that is being called. receiver is the object to use for the "this" value. This will invoke the [[Call]] property on function with the arguments receiver, arg1, ..., argn. The result of invoking the [[Call]] property will be pushed onto the stack.

Runtime exceptions

A TypeError is thrown if function is not a Function.

callmethod

This page last changed on Jun 23, 2010 by john_rau.

Operation

Call a method identified by index in the object's method table.

Format

```
callmethod
index
arg_count
```

Forms

`callmethod = 67 (0x43)`

Stack

..., receiver, arg1, arg2, ..., argn => ..., value

Description

index is a u30 that is the index of the method to invoke on receiver. arg_count is a u30 that is the number of arguments present on the stack. receiver is the object to invoke the method on.

The method at position index on the object receiver, is invoked with the arguments receiver, arg1, ..., argn. The result of the method call is pushed onto the stack.

Runtime exceptions

A TypeError is thrown if receiver is `null` or `undefined`.

An ArgumentError is thrown if the number of arguments does not match the expected number of arguments for the method.

callproperty

Operation

Call a property.

Format

```
callproperty
index
arg_count
```

Forms

```
callproperty = 70 (0x46)
```

Stack

..., obj, [ns], [name], arg1,...,argn => ..., value

Description

arg_count is a u30 that is the number of arguments present on the stack. The number of arguments specified by arg_count are popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

obj is the object to resolve and call the property on.

The property specified by the multiname at index is resolved on the object obj. The [[Call]] property is invoked on the value of the resolved property with the arguments obj, arg1, ..., argn. The result of the call is pushed onto the stack.

Runtime exceptions

A TypeError is thrown if obj is `null` or `undefined` or if the property specified by the multiname is `null` or `undefined`.

An ArgumentError is thrown if the number of arguments does not match the expected number of expected arguments for the method.

callproplex

This page last changed on Jun 23, 2010 by john_rau.

Operation

Call a property.

Format

```
callproplex
index
arg_count
```

Forms

```
callproplex = 76 (0x4c)
```

Stack

..., obj, [ns], [name], arg1,...,argn => ..., value

Description

arg_count is a u30 that is the number of arguments present on the stack. The number of arguments specified by arg_count are popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

obj is the object to resolve and call the property on.

The property specified by the multiname at index is resolved on the object obj. The [[Call]] property is invoked on the value of the resolved property with the arguments null, arg1, ..., argn. The result of the call is pushed onto the stack.

Runtime exceptions

A TypeError is thrown if obj is `null` or `undefined` or if the property specified by the multiname is `null` or `undefined`.

An ArgumentError is thrown if the number of arguments does not match the expected number of expected arguments for the method.

callpropvoid

This page last changed on Jun 23, 2010 by john_rau.

Operation

Call a property, discarding the return value.

Format

```
callpropvoid
index
arg_count
```

Forms

```
callproperty = 79 (0x4f)
```

Stack

..., obj, [ns], [name], arg1,...,argn => ...

Description

arg_count is a u30 that is the number of arguments present on the stack. The number of arguments specified by arg_count are popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

obj is the object to resolve and call the property on.

The property specified by the multiname at index is resolved on the object obj. The [[Call]] property is invoked on the value of the resolved property with the arguments obj, arg1, ..., argn. The result of the call is discarded.

Runtime exceptions

A TypeError is thrown if obj is null or `undefined` or if the property specified by the multiname is null or `undefined`.

An ArgumentError is thrown if the number of arguments does not match the expected number of expected arguments for the method.

callstatic

This page last changed on Jun 23, 2010 by john_rau.

Operation

Call a method identified by index in the abcFile method table.

Format

```
callstatic
index
arg_count
```

Forms

`callstatic = 68 (0x44)`

Stack

..., receiver, arg1, arg2, ..., argn => ..., value

Description

index is a u30 that is the index of the method_info of the method to invoke. arg_count is a u30 that is the number of arguments present on the stack. receiver is the object to invoke the method on.

The method at position index is invoked with the arguments receiver, arg1, ..., argn. The receiver will be used as the "this" value for the method. The result of the method is pushed onto the stack.

Runtime exceptions

A TypeError is thrown if receiver is `null` or `undefined`.

An ArgumentError is thrown if the number of arguments does not match the expected number of arguments for the method.

callsuper

This page last changed on Jun 23, 2010 by john_rau.

Operation

Call a method on a base class.

Format

```
callsuper
index
arg_count
```

Forms

`callsuper = 69 (0x45)`

Stack

..., receiver, [ns/], [name/], arg1,...,argn => ..., value

Description

arg_count is a u30 that is the number of arguments present on the stack. The number of arguments specified by arg_count are popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

receiver is the object to invoke the method on.

The base class of receiver is determined and the method indicated by the multiname is resolved in the declared traits of the base class. The method is invoked with the arguments receiver, arg1, ..., argn. The receiver will be used as the "this" value for the method. The result of the method call is pushed onto the stack.

Runtime exceptions

A TypeError is thrown if receiver is `null` or `undefined`.

An ArgumentError is thrown if the number of arguments does not match the expected number of arguments for the method.

callsupervoid

This page last changed on Jun 23, 2010 by john_rau.

Operation

Call a method on a base class, discarding the return value.

Format

```
callsupervoid
index
arg_count
```

Forms

`callsuper = 78 (0x4e)`

Stack

..., receiver, [ns], [name], arg1, ..., argn => ...

Description

arg_count is a u30 that is the number of arguments present on the stack. The number of arguments specified by arg_count are popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

receiver is the object to invoke the method on.

The base class of receiver is determined and the method indicated by the multiname is resolved in the declared traits of the base class. The method is invoked with the arguments receiver, arg1, ..., argn. The first argument will be used as the "this" value for the method. The result of the method is discarded.

Runtime exceptions

A TypeError is thrown if receiver is `null` or `undefined`.

An ArgumentError is thrown if the number of arguments does not match the expected number of arguments for the method.

checkfilter

This page last changed on Jun 23, 2010 by john_rau.

Operation

Check to make sure an object can have a filter operation performed on it.

Format

```
checkfilter
```

Forms

```
checkfilter = 120 (0x78)
```

Stack

..., value => ..., value

Description

This instruction checks that the top value of the stack can have a filter operation performed on it. If value is of type XML or XMLList then nothing happens. If value is of any other type a TypeError is thrown.

Runtime exceptions

A TypeError is thrown if value is not of type XML or XMLList.

coerce

This page last changed on Jun 23, 2010 by john_rau.

Operation

Coerce a value to a specified type

Format

```
coerce
index
```

Forms

```
coerce = 128 (0x80)
```

Stack

..., value => ..., coercedvalue

Description

index is a u30 that must be an index into the multiname constant pool. The multiname at index must not be a runtime multiname.

The type specified by the multiname is resolved, and value is coerced to that type. The resulting value is pushed onto the stack. If any of value's base classes, or implemented interfaces matches the type specified by the multiname, then the conversion succeeds and the result is pushed onto the stack.

Runtime exceptions

A TypeError is thrown if value cannot be coerced to the specified type.

coerce_a

This page last changed on Jun 23, 2010 by john_rau.

Operation

Coerce a value to the any type.

Format

```
coerce_a
```

Forms

```
coerce_a = 130 (0x82)
```

Stack

..., value => ..., value

Description

Indicates to the verifier that the value on the stack is of the any type (*). Does nothing to value.

coerce_s

This page last changed on Jun 23, 2010 by john_rau.

Operation

Coerce a value to a string.

Format

```
coerce_s
```

Forms

```
coerce_s = 133 (0x85)
```

Stack

..., value => ..., stringvalue

Description

value is popped off of the stack and coerced to a String. If value is `null` or `undefined`, then stringvalue is set to `null`. Otherwise stringvalue is set to the result of the ToString algorithm, as specified in ECMA-262 section 9.8. stringvalue is pushed onto the stack.

Notes

This opcode is very similar to the `convert_s` opcode. The difference is that `convert_s` will convert a `null` or `undefined` value to the string "`null`" or "`undefined`" whereas `coerce_s` will convert those values to the `null` value.

construct

This page last changed on Jun 23, 2010 by john_rau.

Operation

Construct an instance.

Format

```
construct
arg_count
```

Forms

```
construct = 66 (0x42)
```

Stack

..., object, arg1, arg2, ..., argn => ..., value

Description

arg_count is a u30 that is the number of arguments present on the stack. object is the function that is being constructed. This will invoke the [[Construct]] property on object with the given arguments. The new instance generated by invoking [[Construct]] will be pushed onto the stack.

Runtime exceptions

A TypeError is thrown if object does not implement the [[Construct]] property.

This page last changed on Jun 23, 2010 by john_rau.

Operation

Construct a property.

Format

```
constructprop
index
arg_count
```

Forms

```
constructprop = 74 (0x4a)
```

Stack

..., obj, [ns], [name], arg1,...,argn => ..., value

Description

arg_count is a u30 that is the number of arguments present on the stack. The number of arguments specified by arg_count are popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

obj is the object to resolve the multiname in.

The property specified by the multiname at index is resolved on the object obj. The [[Construct]] property is invoked on the value of the resolved property with the arguments obj, arg1, ..., argn. The new instance generated by invoking [[Construct]] will be pushed onto the stack.

Runtime exceptions

A TypeError is thrown if obj is `null` or `undefined`.

A TypeError is thrown if the property specified by the multiname does not implement the [[Construct]] property.

An ArgumentError is thrown if the number of arguments does not match the expected number of expected arguments for the constructor.

constructsuper

This page last changed on Jun 23, 2010 by john_rau.

Operation

Construct an instance of the base class.

Format

```
constructsuper
arg_count
```

Forms

```
construct = 73 (0x49)
```

Stack

..., object, arg1, arg2, ..., argn => ...

Description

arg_count is a u30 that is the number of arguments present on the stack. This will invoke the constructor on the base class of object with the given arguments.

Runtime exceptions

A TypeError is thrown if object is `null` or `undefined`.

convert_b

Operation

Convert a value to a Boolean.

Format

```
convert_b
```

Forms

```
convert_b = 118 (0x76)
```

Stack

..., value => ..., booleanvalue

Description

value is popped off of the stack and converted to a Boolean. The result, booleanvalue, is pushed onto the stack. This uses the ToBoolean algorithm, as described in ECMA-262 section 9.2, to perform the conversion.

convert_d

This page last changed on Jun 23, 2010 by john_rau.

Operation

Convert a value to a double.

Format

```
convert_d
```

Forms

```
convert_d = 117 (0x75)
```

Stack

..., value => ..., doublevalue

Description

value is popped off of the stack and converted to a double. The result, doublevalue, is pushed onto the stack. This uses the ToNumber algorithm, as described in ECMA-262 section 9.3, to perform the conversion.

convert_i

This page last changed on Jun 23, 2010 by john_rau.

Operation

Convert a value to an integer.

Format

```
convert_i
```

Forms

```
convert_i = 115 (0x73)
```

Stack

..., value => ..., intvalue

Description

value is popped off of the stack and converted to an integer. The result, intvalue, is pushed onto the stack. This uses the ToInt32 algorithm, as described in ECMA-262 section 9.5, to perform the conversion.

convert_o

This page last changed on Jun 23, 2010 by john_rau.

Operation

Convert a value to an Object.

Format

convert_o

Forms

`convert_o = 119 (0x77)`

Stack

..., value => ..., value

Description

If value is an Object then nothing happens. Otherwise an exception is thrown.

Runtime exceptions

A TypeError is thrown if value is `null` or `undefined`.

convert_s

This page last changed on Jun 23, 2010 by john_rau.

Operation

Convert a value to a string.

Format

```
convert_s
```

Forms

```
convert_s = 112 (0x70)
```

Stack

..., value => ..., stringvalue

Description

value is popped off of the stack and converted to a string. The result, stringvalue, is pushed onto the stack. This uses the ToString algorithm, as described in ECMA-262 section 9.8

Notes

This is very similar to the `coerce_s` opcode. The difference is that `coerce_s` will not convert a `null` or `undefined` value to the string "`null`" or "`undefined`" whereas `convert_s` will.

convert_u

This page last changed on Jun 23, 2010 by john_rau.

Operation

Convert a value to an unsigned integer.

Format

convert_u

Forms

```
convert_u = 116 (0x74)
```

Stack

..., value => ..., uintvalue

Description

value is popped off of the stack and converted to an unsigned integer. The result, uintvalue, is pushed onto the stack. This uses the ToUint32 algorithm, as described in ECMA-262 section 9.6

debug

This page last changed on Jun 23, 2010 by john_rau.

Operation

Debugging info.

Format

```
debug
debug_type
index
reg
extra
```

Forms

```
debug = 239 (0xef)
```

Stack

... => ...

Description

debug_type is an unsigned byte. If the value of debug_type is `DI_LOCAL` (1), then this is debugging information for a local register.

index is a u30 that must be an index into the string constant pool. The string at index is the name to use for this register.

reg is an unsigned byte and is the index of the register that this is debugging information for.

extra is a u30 that is currently unused.

When debug_type has a value of 1, this tells the debugger the name to display for the register specified by reg. If the debugger is not running, then this instruction does nothing.

debugfile

This page last changed on Jun 23, 2010 by john_rau.

Operation

Debugging line number info.

Format

```
debugfile
index
```

Forms

```
debug = 241 (0xf1)
```

Stack

... => ...

Description

index is a u30 that must be an index into the string constant pool

If the debugger is running, then this instruction sets the current file name in the debugger to the string at position index of the string constant pool. This lets the debugger know which instructions are associated with each source file. The debugger will treat all instructions as occurring in the same file until a new `debugfile` opcode is encountered.

This instruction must occur before any `debugline` opcodes.

debugline

This page last changed on Jun 23, 2010 by john_rau.

Operation

Debugging line number info.

Format

```
debugline
linenum
```

Forms

```
debug = 240 (0xf0)
```

Stack

... => ...

Description

linenum is a u30 that indicates the current line number the debugger should be using for the code currently executing.

If the debugger is running, then this instruction sets the current line number in the debugger. This lets the debugger know which instructions are associated with each line in a source file. The debugger will treat all instructions as occurring on the same line until a new `debugline` opcode is encountered.

declocal

This page last changed on Jun 23, 2010 by john_rau.

Operation

Decrement a local register value.

Format

```
declocal
index
```

Forms

```
declocal = 148 (0x94)
```

Stack

... => ...

Description

index is a u30 that must be an index of a local register. The value of the local register at index is converted to a Number using the ToNumber algorithm (ECMA-262 section 9.3) and then 1 is subtracted from the Number value. The local register at index is then set to the result.

declocal_i

This page last changed on Jun 23, 2010 by john_rau.

Operation

Decrement a local register value.

Format

```
declocal_i
index
```

Forms

```
declocal_i = 195 (0xc3)
```

Stack

... => ...

Description

index is a u30 that must be an index of a local register. The value of the local register at index is converted to an int using the ToInt32 algorithm (ECMA-262 section 9.5) and then 1 is subtracted the int value. The local register at index is then set to the result.

decrement

This page last changed on Jun 23, 2010 by john_rau.

Operation

Decrement a value.

Format

```
decrement
```

Forms

```
decrement = 147 (0x93)
```

Stack

..., value => ..., decrementedvalue

Description

Pop value off of the stack. Convert value to a Number using the ToNumber algorithm (ECMA-262 section 9.3) and then subtract 1 from the Number value. Push the result onto the stack.

decrement_i

This page last changed on Jun 23, 2010 by john_rau.

Operation

Decrement an integer value.

Format

```
decrement_i
```

Forms

```
decrement_i = 193 (0xc1)
```

Stack

..., value => ..., dencrementedvalue

Description

Pop value off of the stack. Convert value to an int using the ToInt32 algorithm (ECMA-262 section 9.5) and then subtract 1 from the int value. Push the result onto the stack.

deleteproperty

This page last changed on Jun 23, 2010 by john_rau.

Operation

Delete a property.

Format

```
deleteproperty
index
```

Forms

`deleteproperty = 106 (0x6a)`

Stack

..., object, [ns/], [name/] => ..., value

Description

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

This will invoke the [[Delete]] method on object with the name specified by the multiname. If object is not dynamic or the property is a fixed property then nothing happens, and `false` is pushed onto the stack. If object is dynamic and the property is not a fixed property, it is removed from object and `true` is pushed onto the stack.

Runtime exceptions

A ReferenceError is thrown if object is `null` or `undefined`.

divide

This page last changed on Jun 23, 2010 by john_rau.

Operation

Divide two values.

Format

```
divide
```

Forms

```
divide = 163 (0xa3)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack, convert value1 and value2 to Number to create value1_number and value2_number. Divide value1_number by value2_number and push the result onto the stack.

dup

This page last changed on Jun 22, 2010 by john_rau.

Operation

Duplicates the top value on the stack.

Format

```
dup
```

Forms

```
dup = 42 (0x2a)
```

Stack

..., value => ..., value, value

Description

Duplicates the top value of the stack, and then pushes the duplicated value onto the stack.

dxns

This page last changed on Jun 23, 2010 by john_rau.

Operation

Sets the default XML namespace.

Format

```
dxns
index
```

Forms

`dxns = 6 (0x06)`

Stack

... => ...

Description

index is a u30 that must be an index into the string constant pool. The string at index is used as the uri for the default XML namespace for this method.

Runtime exceptions

A VerifyError is thrown if `dxns` is used in a method that does not have the `SETS_DXNS` flag set.

dxnslate

This page last changed on Jun 23, 2010 by john_rau.

Operation

Sets the default XML namespace with a value determined at runtime.

Format

```
dxns
```

Forms

```
dxnslate = 7 (0x07)
```

Stack

..., value => ...

Description

The top value on the stack is popped, converted to a string, and that string is used as the uri for the default XML namespace for this method.

Runtime exceptions

A VerifyError is thrown if `dxnslate` is used in a method that does not have the `SETS_DXNS` flag set.

equals

This page last changed on Jun 23, 2010 by john_rau.

Operation

Compare two values.

Format

```
equals
```

Forms

```
equals = 171 (0xab)
```

Stack

..., value1, value2 => ..., result

Description

Pop value1 and value2 off of the stack. Compare the two values using the abstract equality comparison algorithm, as described in ECMA-262 section 11.9.3 and extended in ECMA-347 section 11.5.1. Push the resulting Boolean value onto the stack.

esc_xattr

This page last changed on Jun 23, 2010 by john_rau.

Operation

Escape an xml attribute.

Format

```
esc_xattr
```

Forms

```
esc_xattr = 114 (0x72)
```

Stack

..., value => ..., stringvalue

Description

value is popped off of the stack and converted to a string. The result, stringvalue, is pushed onto the stack. This uses the EscapeAttributeValue algorithm as described in the E4X specification, ECMA-357 section 10.2.1.2, to perform the conversion.

esc_xelem

This page last changed on Jun 23, 2010 by john_rau.

Operation

Escape an xml element.

Format

```
esc_xelem
```

Forms

```
esc_xelem = 113 (0x71)
```

Stack

..., value => ..., stringvalue

Description

value is popped off of the stack and converted to a string. The result, stringvalue, is pushed onto the stack. This uses the ToXmlString algorithm as described in the E4X specification, ECMA-357 section 10.2, to perform the conversion.

findproperty

This page last changed on Jun 23, 2010 by john_rau.

Operation

Search the scope stack for a property.

Format

```
findproperty
index
```

Forms

```
findproperty = 94 (0x5e)
```

Stack

..., [ns], [name] => ..., obj

Description

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

This searches the scope stack, and then the saved scope in the current method closure, for a property with the name specified by the multiname at index.

If any of the objects searched is a `with` scope, its declared and dynamic properties will be searched for a match. Otherwise only the declared traits of a scope will be searched. The global object will have its declared traits, dynamic properties, and prototype chain searched.

If the property is resolved then the object it was resolved in is pushed onto the stack. If the property is unresolved in all objects on the scope stack then the global object is pushed onto the stack.

Notes

Functions save the scope stack when they are created, and this saved scope stack is searched if no match is found in the current scope stack.

Objects for the `with` statement are pushed onto the scope stack with the `pushwith` instruction.

findpropstrict

This page last changed on Jun 23, 2010 by john_rau.

Operation

Find a property.

Format

```
findpropstrict
index
```

Forms

```
findpropstrict = 93 (0x5d)
```

Stack

..., [ns], [name] => ..., obj

Description

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

This searches the scope stack, and then the saved scope in the method closure, for a property with the name specified by the multiname at index.

If any of the objects searched is a `with` scope, its declared and dynamic properties will be searched for a match. Otherwise only the declared traits of a scope will be searched. The global object will have its declared traits, dynamic properties, and prototype chain searched.

If the property is resolved then the object it was resolved in is pushed onto the stack. If the property is unresolved in all objects on the scope stack then an exception is thrown.

Runtime exceptions

A ReferenceError is thrown if the property is not resolved in any object on the scope stack.

Notes

Functions save the scope stack when they are created, and this saved scope stack is searched if no match is found in the current scope stack.

Objects for the `with` statement are pushed onto the scope stack with the `pushwith` instruction.

getdescendants

This page last changed on Jun 23, 2010 by john_rau.

Operation

Get descendants.

Format

```
getdescendants
index
```

Forms

```
getdescendants = 89 (0x59)
```

Stack

..., obj, [ns], [name] => ..., value

Description

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

obj is the object to find the descendants in. This will invoke the [[Descendants]] property on obj with the multiname specified by index. For a description of the [[Descendants]] operator, see the E4X spec (ECMA-357) sections 9.1.1.8 (for the XML type) and 9.2.1.8 (for the XMLList type).

Runtime exceptions

A TypeError is thrown if obj is not of type XML or XMLList.

getglobalscope

This page last changed on Jun 22, 2010 by john_rau.

Operation

Gets the global scope.

Format

```
getglobalscope
```

Forms

```
getglobalscope = 100 (0x64)
```

Stack

... => ..., obj

Description

Gets the global scope object from the scope stack, and pushes it onto the stack. The global scope object is the object at the bottom of the scope stack.

getglobalslot

This page last changed on Jun 23, 2010 by john_rau.

Operation

Get the value of a slot on the global scope.

Format

```
getglobalslot
slotindex
```

Forms

```
getglobalslot = 110 (0x6e)
```

Stack

... => ..., value

Description

slotindex is a u30 that must be an index of a slot on the global scope. The slotindex must be greater than 0 and less than or equal to the total number of slots the global scope has.

This will retrieve the value stored in the slot at slotindex of the global scope. This value is pushed onto the stack.

getlex

Operation

Find and get a property.

Format

```
getlex
index
```

Forms

```
getlex = 96 (0x60)
```

Stack

... => ..., obj

Description

index is a u30 that must be an index into the multiname constant pool. The multiname at index must not be a runtime multiname, so there are never any optional namespace or name values on the stack.

This is the equivalent of doing a `findpropstict` followed by a `getproperty`. It will find the object on the scope stack that contains the property, and then will get the value from that object. See "Resolving multinames" on page 10.

Runtime exceptions

A ReferenceError is thrown if the property is unresolved in all of the objects on the scope stack.

getlocal

This page last changed on Jun 23, 2010 by john_rau.

Operation

Get a local register.

Format

```
getlocal
index
```

Forms

```
getlocal = 98 (0x62)
```

Stack

... => ..., value

Description

index is a u30 that must be an index of a local register. The value of that register is pushed onto the stack.

getlocal_n

Operation

Get a local register.

Format

```
getlocal_<n>
```

Forms

```
getlocal_0 = 208 (0xd0)
getlocal_1 = 209 (0xd1)
getlocal_2 = 210 (0xd2)
getlocal_3 = 211 (0xd3)
```

Stack

... => ..., value

Description

<n> is the index of a local register. The value of that register is pushed onto the stack.

getproperty

getproperty

Operation

Get a property.

Format

```
getproperty
index
```

Forms

```
getproperty = 102 (0x66)
```

Stack

..., object, [ns], [name] => ..., value

Description

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

The property with the name specified by the multiname will be resolved in object, and the value of that property will be pushed onto the stack. If the property is unresolved, `undefined` is pushed onto the stack. See "Resolving multinames" on page 10.

getscopeobject

This page last changed on Jun 23, 2010 by john_rau.

getscopeobject

Operation

Get a scope object.

Format

```
getscopeobject
index
```

Forms

```
getscopeobject = 101 (0x65)
```

Stack

... => ..., scope

Description

index is an unsigned byte that specifies the index of the scope object to retrieve from the local scope stack. index must be less than the current depth of the scope stack. The scope at that index is retrieved and pushed onto the stack. The scope at the top of the stack is at index scope_depth-1, and the scope at the bottom of the stack is index 0.

Notes

The indexing of elements on the local scope stack is the reverse of the indexing of elements on the local operand stack.

getslot

Operation

Get the value of a slot.

Format

```
getslot
slotindex
```

Forms

```
getslot = 108 (0x6c)
```

Stack

..., obj => ..., value

Description

slotindex is a u30 that must be an index of a slot on obj. slotindex must be less than the total number of slots obj has.

This will retrieve the value stored in the slot at slotindex on obj. This value is pushed onto the stack.

Runtime exceptions

A TypeError is thrown if obj is `null` or `undefined`.

getsuper

This page last changed on Jun 23, 2010 by john_rau.

Operation

Gets a property from a base class.

Format

```
getsuper
index
```

Forms

`getsuper = 4 (0x04)`

Stack

..., obj, [ns], [name] => ..., value

Description

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime

Once the multiname is constructed, the base class of obj is determined and the multiname is resolved in the declared traits of the base class. The value of the resolved property is pushed onto the stack. See "Resolving multinames" on page 10.

Runtime exceptions

A TypeError is thrown if obj is `null` or `undefined`.

A ReferenceError is thrown if the property is unresolved, or if the property is write-only.

greaterequals

This page last changed on Jun 23, 2010 by john_rau.

Operation

Determine if one value is greater than or equal to another.

Format

```
greaterthan
```

Forms

```
greaterthan = 175 (0xb0)
```

Stack

..., value1, value2 => ..., result

Description

Pop value1 and value2 off of the stack. Compute value1 < value2 using the Abstract Relational Comparison Algorithm, as described in ECMA-262 section 11.8.5. If the result of the comparison is `false`, push `true` onto the stack. Otherwise push `false` onto the stack.

greaterthan

This page last changed on Jun 23, 2010 by john_rau.

Operation

Determine if one value is greater than another.

Format

```
greaterthan
```

Forms

```
greaterthan = 175 (0xaf)
```

Stack

..., value1, value2 => ..., result

Description

Pop value1 and value2 off of the stack. Compute value2 < value1 using the Abstract Relational Comparison Algorithm as described in ECMA-262 section 11.8.5. If the result of the comparison is `true`, push `true` onto the stack. Otherwise push `false` onto the stack.

hasnext

This page last changed on Jun 23, 2010 by john_rau.

Operation

Determine if the given object has any more properties.

Format

```
hasnext
```

Forms

```
hasnext = 31(0x1f)
```

Stack

..., obj, cur_index => ..., next_index

Description

cur_index and obj are popped off of the stack. cur_index must be of type int. Get the index of the next property after the property at cur_index. If there are no more properties, then the result is 0. The result is pushed onto the stack.

hasnext2

This page last changed on Jun 23, 2010 by john_rau.

Operation

Determine if the given object has any more properties.

Format

```
hasnext2
object_reg
index_reg
```

Forms

`hasnext2 = 50 (0x32)`

Stack

..., => ..., value

Description

object_reg and index_reg are u30s that must be indexes to a local register. The value of the register at position object_reg is the object that is being enumerated and is assigned to obj. The value of the register at position index_reg must be of type int, and that value is assigned to cur_index.

Get the index of the next property after the property located at index cur_index on object obj. If there are no more properties on obj, then obj is set to the next object on the prototype chain of obj, and cur_index is set to the first index of that object. If there are no more objects on the prototype chain and there are no more properties on obj, then obj is set to `null`, and cur_index is set to 0.

The register at position object_reg is set to the value of obj, and the register at position index_reg is set to the value of cur_index.

If index is not 0, then push `true`. Otherwise push `false`.

Notes

`hasnext2` works by reference. Each time it is executed it changes the values of local registers rather than simply returning a new value. This is because the object being enumerated can change when it is necessary to walk up the prototype chain to find more properties. This is different from how `hasnext` works, though the two may seem similar due to the similar names.

ifeq

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is equal to the second value.

Format

```
ifeq
offset
```

Forms

```
ifeq = 19 (0x13)
```

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is equal to value2.

Compute value1 == value2 using the abstract equality comparison algorithm in ECMA-262 section 11.9.3 and ECMA-347 section 11.5.1. If the result of the comparison is `true`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

iffalse

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if false.

Format

```
iffalse
offset
```

Forms

```
iffalse = 18 (0x12)
```

Stack

..., value => ...

Description

offset is an s24 that is the number of bytes to jump.

Pop value off the stack and convert it to a Boolean. If the converted value is `false`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

ifge

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is greater than or equal to the second value.

Format

```
ifge
offset
```

Forms

```
ifge = 24 (0x18)
```

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is greater than or equal to value2.

Compute value1 < value2 using the abstract relational comparison algorithm in ECMA-262 section 11.8.5. If the result of the comparison is `false`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

ifgt

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is greater than the second value.

Format

```
ifgt
offset
```

Forms

```
ifgt = 23 (0x17)
```

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is greater than or equal to value2.

Compute value2 < value1 using the abstract relational comparison algorithm in ECMA-262 section 11.8.5. If the result of the comparison is `true`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

ifle

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is less than or equal to the second value.

Format

```
ifle
offset
```

Forms

```
ifle = 22 (0x16)
```

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is less than or equal to value2.

Compute value2 < value1 using the abstract relational comparison algorithm in ECMA-262 section 11.8.5. If the result of the comparison is `false`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is less than the second value.

Format

```
iflt
offset
```

Forms

```
iflt = 21 (0x15)
```

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is less than value2.

Compute value1 < value2 using the abstract relational comparison algorithm in ECMA-262 section 11.8.5. If the result of the comparison is `true`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

ifne

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is not equal to the second value.

Format

```
ifne
offset
```

Forms

`ifne = 20 (0x14)`

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is not equal to value2.

Compute value1 == value2 using the abstract equality comparison algorithm in ECMA-262 section 11.9.3 and ECMA-347 Section 11.5.1. If the result of the comparison is `false`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

ifnge

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is not greater than or equal to the second value.

Format

```
ifnge
offset
```

Forms

`ifnge = 15 (0x0f)`

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is not greater than or equal to value2.

Compute value1 < value2 using the abstract relational comparison algorithm in ECMA-262 section 11.8.5. If the result of the comparison is not `false`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

Notes

This appears to have the same effect as `iflt`, however, their handling of `NaN` is different. If either of the compared values is `NaN` then the comparison value1 < value2 will return `undefined`. In that case `ifnge` will branch (`undefined` is not `false`), but `iflt` will not branch.

ifngt

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is not greater than the second value.

Format

```
ifngt
offset
```

Forms

`ifngt = 14 (0x0e)`

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is not greater than or value2.

Compute value2 < value1 using the abstract relational comparison algorithm in ECMA-262 section 11.8.5. If the result of the comparison is not `true`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

Notes

This appears to have the same effect as `ifle`, however, their handling of `NaN` is different. If either of the compared values is `NaN` then the comparison value2 < value1 will return `undefined`. In that case `ifngt` will branch (`undefined` is not `true`), but `ifle` will not branch.

ifnle

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is not less than or equal to the second value.

Format

```
ifnle
offset
```

Forms

`ifnle = 13 (0x0d)`

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is not less than value2.

Compute value1 < value2 using the abstract relational comparison algorithm in ECMA-262 section 11.8.5. If the result of the comparison is `false`, then jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

Notes

This appears to have the same effect as `ifge`, however, their handling of `NaN` is different. If either of the compared values is `NaN` then the comparison value1 < value2 will return `undefined`. In that case `ifnlt` will branch (`undefined` is not `true`), but `ifge` will not branch.

ifnlt

This page last changed on Jun 22, 2010 by john_rau.

Operation

Branch if the first value is not less than the second value.

Format

```
ifnlt
offset
```

Forms

```
ifnlt = 12 (0x0c)
```

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is not less than value2.

Compute value1 < value2 using the abstract relational comparison algorithm in ECMA-262 section 11.8.5. If the result of the comparison is false, then jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

Notes

This appears to have the same effect as ifge, however, their handling of NaN is different. If either of the compared values is NaN then the comparison value1 < value2 will return undefined. In that case ifnlt will branch (undefined is not true), but ifge will not branch.

ifstricteq

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if the first value is equal to the second value.

Format

```
ifstricteq
offset
```

Forms

```
ifstricteq = 24 (0x19)
```

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is equal to value2.

Compute value1 === value2 using the strict equality comparison algorithm in ECMA-262 section 11.9.6. If the result of the comparison is `true`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

ifstrictne

Operation

Branch if the first value is not equal to the second value.

Format

```
ifstrictne
offset
```

Forms

```
ifstrictne = 25 (0x1a)
```

Stack

..., value1, value2 => ...

Description

offset is an s24 that is the number of bytes to jump if value1 is not equal to value2.

Compute value1 === value2 using the strict equality comparison algorithm in ECMA-262 section 11.9.6. If the result of the comparison is `false`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

iftrue

This page last changed on Jun 23, 2010 by john_rau.

Operation

Branch if true.

Format

```
iftrue
offset
```

Forms

```
iftrue = 17 (0x11)
```

Stack

..., value => ...

Description

offset is an s24 that is the number of bytes to jump.

Pop value off the stack and convert it to a Boolean. If the converted value is `true`, jump the number of bytes indicated by offset. Otherwise continue executing code from this point.

in

This page last changed on Jun 23, 2010 by john_rau.

Operation

Determine whether an object has a named property.

Format

```
in
```

Forms

```
in = 180 (0xb4)
```

Stack

..., name, obj => ..., result

Description

name is converted to a String, and is looked up in obj. If no property is found, then the prototype chain is searched by calling [[HasProperty]] on the prototype of obj. If the property is found result is `true`. Otherwise result is `false`. Push result onto the stack.

inclocal

This page last changed on Jun 23, 2010 by john_rau.

Operation

Increment a local register value.

Format

```
inclocal
index
```

Forms

```
inclocal = 146 (0x92)
```

Stack

... => ...

Description

index is a u30 that must be an index of a local register. The value of the local register at index is converted to a Number using the ToNumber algorithm (ECMA-262 section 9.3) and then 1 is added to the Number value. The local register at index is then set to the result.

inclocal_i

This page last changed on Jun 23, 2010 by john_rau.

Operation

Increment a local register value.

Format

```
inclocal_i
index
```

Forms

```
inclocal_i = 194 (0xc2)
```

Stack

... => ...

Description

index is a u30 that must be an index of a local register. The value of the local register at index is converted to an int using the ToInt32 algorithm (ECMA-262 section 9.5) and then 1 is added to the int value. The local register at index is then set to the result.

increment

This page last changed on Jun 23, 2010 by john_rau.

Operation

Increment a value.

Format

```
increment
```

Forms

```
increment = 145 (0x91)
```

Stack

..., value => ..., incrementedvalue

Description

Pop value off of the stack. Convert value to a Number using the ToNumber algorithm (ECMA-262 section 9.3) and then add 1 to the Number value. Push the result onto the stack.

increment_i

This page last changed on Jun 23, 2010 by john_rau.

Operation

Increment an integer value.

Format

```
increment_i
```

Forms

```
increment_i = 192 (0xc0)
```

Stack

..., value => ..., incrementedvalue

Description

Pop value off of the stack. Convert value to an int using the ToInt32 algorithm (ECMA-262 section 9.5) and then add 1 to the int value. Push the result onto the stack.

initproperty

This page last changed on Jun 23, 2010 by john_rau.

Operation

Initialize a property.

Format

```
initproperty
index
```

Forms

```
initproperty = 104 (0x68)
```

Stack

..., object, [ns], [name], value => ...

Description

value is the value that the property will be set to. value is popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

The property with the name specified by the multiname will be resolved in object, and will be set to value. This is used to initialize properties in the initializer method. When used in an initializer method it is able to set the value of `const` properties.

Runtime exceptions

A TypeError is thrown if object is `null` or `undefined`.

A ReferenceError is thrown if the property is not found and object is not dynamic, or if the instruction is used to set a `const` property outside an initializer method.

instanceof

This page last changed on Jun 23, 2010 by john_rau.

Operation

Check the prototype chain of an object for the existence of a type.

Format

```
instanceof
```

Forms

```
instanceof = 177 (0xb1)
```

Stack

..., value, type => ..., result

Description

Pop value and type off of the stack. If value is `null` result is `false`. Walk up the prototype chain of value looking for type. If type is present anywhere on the prototype, result is `true`. If type is not found on the prototype chain, result is `false`. Push result onto the stack. See ECMA-262 section 11.8.6 for a further description.

Runtime exceptions

A TypeError is thrown if type is not an Object.

istype

This page last changed on Jun 23, 2010 by john_rau.

Operation

Check whether an Object is of a certain type.

Format

```
istype
index
```

Forms

```
istype = 178 (0xb2)
```

Stack

..., value => ..., result

Description

index is a u30 that must be an index into the multiname constant pool. The multiname at index must not be a runtime multiname.

Resolve the type specified by the multiname. Let indexType refer to that type. Compute the type of value, and let valueType refer to that type. If valueType is the same as indexType, result is `true`. If indexType is a base type of valueType, or an implemented interface of valueType, then result is `true`. Otherwise result is set to `false`. Push result onto the stack.

istypelate

This page last changed on Jun 23, 2010 by john_rau.

Operation

Check whether an Object is of a certain type.

Format

```
istypelate
```

Forms

```
istypelate = 179 (0xb3)
```

Stack

..., value, type => ..., result

Description

Compute the type of value, and let valueType refer to that type. If valueType is the same as type, result is `true`. If type is a base type of valueType, or an implemented interface of valueType, then result is `true`. Otherwise result is set to `false`. Push result onto the stack.

Runtime exceptions

A TypeError is thrown if type is not a Class.

jump

This page last changed on Jun 23, 2010 by john_rau.

Operation

Unconditional branch.

Format

```
jump
offset
```

Forms

```
jump = 16 (0x10)
```

Stack

... => ...

Description

offset is an s24 that is the number of bytes to jump. Jump the number of bytes indicated by offset and resume execution there.

kill

This page last changed on Jun 23, 2010 by john_rau.

Operation

Kills a local register.

Format

```
kill
index
```

Forms

```
kill = 8 (0x08)
```

Stack

... => ...

Description

index is a u30 that must be an index of a local register. The local register at index is killed. It is killed by setting its value to `undefined`.

Notes

This is usually used so that different jumps to the same location will have the same types in the local registers. The verifier ensures that all paths to a location have compatible values in the local registers, if not a VerifyError occurs. This can be used to kill temporary values that were stored in local registers before a jump so that no VerifyError occurs.

label

Operation

Do nothing.

Format

```
label
```

Forms

```
label = 9 (0x09)
```

Stack

... => ...

Description

Do nothing. Used to indicate that this location is the target of a branch.

Notes

This is usually used to indicate the target of a backwards branch. The `label` opcode will prevent the verifier from thinking that the code after the `label` is unreachable.

lessequals

This page last changed on Jun 23, 2010 by john_rau.

Operation

Determine if one value is less than or equal to another.

Format

```
lessequals
```

Forms

```
lessequals = 174 (0xae)
```

Stack

..., value1, value2 => ..., result

Description

Pop value1 and value2 off of the stack. Compute value2 < value1 using the Abstract Relational Comparison Algorithm as described in ECMA-262 section 11.8.5. If the result of the comparison is `false`, push `true` onto the stack. Otherwise push `false` onto the stack.

lessthan

This page last changed on Jun 23, 2010 by john_rau.

Operation

Determine if one value is less than another.

Format

```
lessthan
```

Forms

```
lessthan = 173 (0xad)
```

Stack

..., value1, value2 => ..., result

Description

Pop value1 and value2 off of the stack. Compute value1 < value2 using the Abstract Relational Comparison Algorithm as described in ECMA-262 section 11.8.5. If the result of the comparison is `true`, then push `true` onto the stack. Otherwise push `false` onto the stack.

lf32

Operation

Load single-precision floating point indirect.

Format

```
lf32
```

Forms

```
lf32 = 56 (0x38)
```

Stack

..., offset => ..., result

Description

Pop offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Push the 32-bit floating point number at that offset in the ByteArray onto the stack.

lf64

Operation

Load double-precision floating point indirect.

Format

```
lf64
```

Forms

```
lf64 = 53 (0x35)
```

Stack

..., offset => ..., result

Description

Pop offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Push the double value at that offset in the ByteArray onto the stack.

li16

Operation

Load short indirect.

Format

```
li16
```

Forms

```
li16 = 54 (0x36)
```

Stack

..., offset => ..., result

Description

Pop offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Push the 16-bit value at that offset in the ByteArray onto the stack.

li32

This page last changed on Jun 23, 2010 by john_rau.

Operation

Load int indirect.

Format

```
li32
```

Forms

```
li32 = 55 (0x37)
```

Stack

..., offset => ..., result

Description

Pop offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Push the 32-bit integer value at that offset in the ByteArray onto the stack.

li8

Operation

Load byte indirect.

Format

```
li8
```

Forms

```
li8 = 53 (0x35)
```

Stack

..., offset => ..., result

Description

Pop offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current domainMemory property. Push the byte at that offset in the ByteArray onto the stack.

lookupswitch

This page last changed on Jun 23, 2010 by john_rau.

Operation

Jump to different locations based on an index.

Format

```
lookupswitch
default_offset
case_count
case_offsets...
```

Forms

```
lookupswitch = 27(0x1b)
```

Stack

..., index => ...

Description

default_offset is an s24 that is the offset to jump, in bytes, for the default case. case_offsets are each an s24 that is the offset to jump for a particular index. There are case_count+1 case offsets. case_count is a u30.

index is popped off of the stack and must be of type int. If index is less than zero or greater than case_count, the target is calculated by adding default_offset to the base location. Otherwise the target is calculated by adding the case_offset at position index to the base location. Execution continues from the target location.

The base location is the address of the `lookupswitch` instruction itself.

Notes

Other control flow instructions take the base location to be the address of the following instruction.

lshift

This page last changed on Jun 23, 2010 by john_rau.

Operation

Bitwise left shift.

Format

```
lshift
```

Forms

```
lshift = 165 (0xa5)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack; convert value1 to an int to create value1_int ; and convert value2 to a uint to create value2_uint. Left shift value1_int by the result of value2_uint & 0x1F (leaving only the 5 least significant bits of value2_uint), and push the result onto the stack. See ECMA-262 section 11.7.1.

modulo

This page last changed on Jun 23, 2010 by john_rau.

Operation

Perform modulo division on two values.

Format

```
modulo
```

Forms

```
modulo = 164 (0xa4)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack, convert value1 and value2 to Number to create value1_number and value2_number. Perform value1_number mod value2_number and push the result onto the stack.

multiply

This page last changed on Jun 23, 2010 by john_rau.

Operation

Multiply two values.

Format

```
multiply
```

Forms

```
multiply = 162 (0xa2)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack, convert value1 and value2 to Number to create value1_number and value2_number. Multiply value1_number by value2_number and push the result onto the stack.

multiply_i

multiply_i

Operation

Multiply two integer values.

Format

```
multiply_i
```

Forms

```
multiply_i = 199 (0xc7)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack, convert value1 and value2 to int to create value1_int and value2_int. Multiply value1_int by value2_int and push the result onto the stack.

negate

This page last changed on Jun 23, 2010 by john_rau.

Operation

Negate a value.

Format

```
negate
```

Forms

```
negate = 144 (0x90)
```

Stack

..., value => ..., -value

Description

Pop value off of the stack. Convert value to a Number using the ToNumber algorithm (ECMA-262 section 9.3) and then negate the Number value. Push the result onto the stack.

negate_i

This page last changed on Jun 23, 2010 by john_rau.

Operation

Negate an integer value.

Format

```
negate_i
```

Forms

```
negate_i = 196 (0xc4)
```

Stack

..., value => ..., -value

Description

Pop value off of the stack. Convert value to an int using the ToInt32 algorithm (ECMA-262 section 9.5) and then negate the int value. Push the result onto the stack.

newactivation

This page last changed on Jun 23, 2010 by john_rau.

Operation

Create a new activation object.

Format

```
newactivation
```

Forms

```
newactivation = 87 (0x57)
```

Stack

... => ..., newactivation

Description

Creates a new activation object, newactivation, and pushes it onto the stack. Can only be used in methods that have the `NEED_ACTIVATION` flag set in their `MethodInfo` entry.

newarray

Operation

Create a new array.

Format

```
newarray
arg_count
```

Forms

```
newarray = 86 (0x56)
```

Stack

..., value1, value2, ..., valueN => ..., newarray

Description

arg_count is a u30 that is the number of entries that will be created in the new array. There will be a total of arg_count values on the stack.

A new value of type Array is created and assigned to newarray. The values on the stack will be assigned to the entries of the array, so newarray[0] = value1, newarray[1] = value2, ..., newarray[N-1] = valueN. newarray is then pushed onto the stack.

newcatch

This page last changed on Jun 23, 2010 by john_rau.

Operation

Create a new catch scope.

Format

```
newcatch
index
```

Forms

`newcatch = 90 (0x5a)`

Stack

... => ..., catchscope

Description

index is a u30 that must be an index of an `exception_info` structure for this method.

This instruction creates a new object to serve as the scope object for the `catch` block for the exception referenced by index. This new scope is pushed onto the operand stack.

newclass

This page last changed on Jun 23, 2010 by john_rau.

Operation

Create a new class.

Format

```
newclass
index
```

Forms

```
newclass = 88 (0x58)
```

Stack

..., basetype => ..., newclass

Description

index is a u30 that is an index of the `ClassInfo` that is to be created. basetype must be the base class of the class being created, or `null` if there is no base class.

The class that is represented by the ClassInfo at position index of the `ClassInfo` entries is created with the given basetype as the base class. This will run the static initializer function for the class. The new class object, newclass, will be pushed onto the stack.

When this instruction is executed, the scope stack must contain all the scopes of all base classes, as the scope stack is saved by the created `ClassClosure`.

newfunction

This page last changed on Jun 23, 2010 by john_rau.

Operation

Create a new function object.

Format

```
newfunction
index
```

Forms

`newfunction = 64 (0x40)`

Stack

... => ..., function_obj

Description

index is a u30 that must be an index of a `method_info`. A new function object is created from that `method_info` and pushed onto the stack. For a description of creating a new function object, see ECMA-262 section 13.2.

When creating the new function object the scope stack used is the current scope stack when this instruction is executed, and the body is the `method_body` entry that references the specified `method_info` entry.

newobject

Operation

Create a new object.

Format

```
newobject
arg_count
```

Forms

```
newobject = 85 (0x55)
```

Stack

..., name1, value1, name2, value2,...,nameN, valueN => ..., newobj

Description

arg_count is a u30 that is the number of properties that will be created in newobj. There will be a total of arg_count name values on the stack, which will be of type String (name1 to nameN). There will be an equal number of values on the stack, which can be of any type, and will be the initial values for the properties

A new value of type Object is created and assigned to newobj. The properties specified on the stack will be dynamically added to newobj. The names of the properties will be name1, name2,..., nameN and these properties will be set to the corresponding values (value1, value2,..., valueN). newobj is then pushed onto the stack.

nextname

This page last changed on Jun 23, 2010 by john_rau.

Operation

Get the name of the next property when iterating over an object.

Format

```
nextname
```

Forms

```
nextname = 30(0x1e)
```

Stack

..., obj, index => ..., name

Description

index and obj are popped off of the stack. index must be a value of type int. Gets the name of the property that is at position index + 1 on the object obj, and pushes it onto the stack.

Notes

index will usually be the result of executing hasnext on obj.

nextvalue

This page last changed on Jun 23, 2010 by john_rau.

Operation

Get the name of the next property when iterating over an object.

Format

```
nextvalue
```

Forms

```
nextvalue = 35(0x23)
```

Stack

..., obj, index => ..., value

Description

index and obj are popped off of the stack. index must be of type int. Get the value of the property that is at position index + 1 on the object obj, and pushes it onto the stack.

Notes

Index will usually be the result of executing hasnext on obj.

nop

This page last changed on Jun 22, 2010 by john_rau.

Operation

Do nothing.

Format

```
nop
```

Forms

```
nop = 2 (0x02)
```

Stack

... => ...

Description

Do nothing.

not

This page last changed on Jun 23, 2010 by john_rau.

Operation

Boolean negation.

Format

```
not
```

Forms

```
not = 150 (0x96)
```

Stack

..., value => ..., !value

Description

Pop value off of the stack. Convert value to a Boolean using the ToBoolean algorithm (ECMA-262 section 9.2) and then negate the Boolean value. Push the result onto the stack.

pop

This page last changed on Jun 22, 2010 by john_rau.

Operation

Pop the top value from the stack.

Format

```
pop
```

Forms

```
pop = 41 (0x29)
```

Stack

..., value => ...

Description

Pops the top value from the stack and discards it.

popscope

This page last changed on Jun 22, 2010 by john_rau.

Operation

Pop a scope off of the scope stack

Format

```
popscope
```

Forms

```
popscope = 29(0x1d)
```

Stack

... => ...

Description

Pop the top scope off of the scope stack and discards it.

pushbyte

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push a byte value.

Format

```
pushbyte
byte_value
```

Forms

```
pushbyte = 36 (0x24)
```

Stack

... => ..., value

Description

byte_value is an unsigned byte . The byte_value is promoted to an int , and the result is pushed onto the stack.

pushdouble

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push a double value onto the stack.

Format

```
pushdouble
index
```

Forms

`pushdouble = 46 (0x2f)`

Stack

... => ..., value

Description

index is a u30 that must be an index into the `double` constant pool. The double value at index in the `double` constant pool is pushed onto the stack.

pushfalse

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push `false`.

Format

```
pushfalse
```

Forms

```
pushfalse = 39 (0x27)
```

Stack

... => ..., false

Description

Push the `false` value onto the stack.

pushint

Operation

Push an int value onto the stack.

Format

```
pushint
index
```

Forms

```
pushint = 45 (0x2d)
```

Stack

... => ..., value

Description

index is a u30 that must be an index into the integer constant pool. The int value at index in the integer constant pool is pushed onto the stack.

pushnamespace

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push a namespace.

Format

```
pushnamespace
index
```

Forms

```
pushnamespace = 49 (0x31)
```

Stack

... => ..., namespace

Description

index is a u30 that must be an index into the namespace constant pool. The namespace value at index in the namespace constant pool is pushed onto the stack.

pushnan

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push `NaN`.

Format

```
pushnan
```

Forms

`pushnan` = 40 (0x28)

Stack

... => ..., *NaN*

Description

Push the value `NaN` onto the stack.

pushnull

Operation

Push {{nul}}l.

Format

```
pushnull
```

Forms

```
pushnull = 32 (0x20)
```

Stack

... => ..., *null*

Description

Push the `null` value onto the stack.

pushscope

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push an object onto the scope stack.

Format

```
pushscope
```

Forms

```
pushscope = 48 (0x30)
```

Stack

..., value => ...

Description

Pop value off of the stack. Push value onto the scope stack.

Runtime exceptions

A TypeError is thrown if value is `null` or `undefined`.

pushshort

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push a short value.

Format

```
pushshort
value
```

Forms

```
pushshort = 37 (0x25)
```

Stack

... => ..., value

Description

value is a u30, to be interpreted as a two's complement signed value. The value is pushed onto the stack.

pushstring

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push a string value onto the stack.

Format

```
pushstring
index
```

Forms

```
pushstring = 44 (0x2c)
```

Stack

... => ..., value

Description

index is a u30 that must be an index into the string constant pool. The string value at index in the string constant pool is pushed onto the stack.

pushtrue

Operation

Push true.

Format

```
pushtrue
```

Forms

```
pushtrue = 38 (0x26)
```

Stack

... => ..., *true*

Description

Push the `true` value onto the stack.

pushuint

Operation

Push an unsigned int value onto the stack.

Format

```
pushuint
index
```

Forms

```
pushuint = 46 (0x2e)
```

Stack

... => ..., value

Description

index is a u30 that must be an index into the unsigned integer constant pool. The value at index in the unsigned integer constant pool is pushed onto the stack.

pushundefined

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push `undefined`.

Format

```
pushundefined
```

Forms

```
pushundefined = 33 (0x21)
```

Stack

... => ..., *undefined*

Description

Push the `undefined` value onto the stack.

pushwith

This page last changed on Jun 23, 2010 by john_rau.

Operation

Push a with scope onto the scope stack

Format

```
pushwith
```

Forms

```
pushwith = 28 (0x1c)
```

Stack

..., scope_obj => ...

Description

scope_obj is popped off of the stack, and the object is pushed onto the scope stack. scope_obj can be of any type.

Runtime exceptions

A TypeError is thrown if scope_obj is `null` or `undefined`.

returnvalue

Operation

Return a value from a method.

Format

```
returnvalue
```

Forms

```
returnvalue = 72 (0x48)
```

Stack

..., return_value => ...

Description

Return from the currently executing method. This returns the top value on the stack. return_value is popped off of the stack, and coerced to the expected return type of the method. The coerced value is what is actually returned from the method.

Runtime exceptions

A TypeError is thrown if return_value cannot be coerced to the expected return type of the executing method.

returnvoid

This page last changed on Jun 23, 2010 by john_rau.

Operation

Return from a method.

Format

```
returnvoid
```

Forms

```
returnvoid = 71 (0x47)
```

Stack

... => ...

Description

Return from the currently executing method. This returns the value `undefined`. If the method has a return type, then `undefined` is coerced to that type and then returned.

rshift

This page last changed on Jun 23, 2010 by john_rau.

Operation

Signed bitwise right shift.

Format

```
rshift
```

Forms

```
rshift = 166 (0xa6)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack, convert value1 to an int to create value1_int and convert value2 to a uint to create value2_uint. Right shift value1_int by the result of value2_uint & 0x1F (leaving only the 5 least significant bits of value2_uint), and push the result onto the stack. The right shift is sign extended, resulting in a signed 32-bit integer. See ECMA-262 section 11.7.2

setglobalslot

This page last changed on Jun 23, 2010 by john_rau.

Operation

Set the value of a slot on the global scope.

Format

```
setglobalslot
slotindex
```

Forms

```
setglobalslot = 111 (0x6f)
```

Stack

..., value => ...

Description

slotindex is a u30 that must be an index of a slot on the global scope. The slotindex must be greater than zero and less than or equal to the total number of slots the global scope has.

This instruction will set the value of the slot at slotindex of the global scope to value. value is first coerced to the type of the slot indicated by slotindex.

setlocal

This page last changed on Jun 23, 2010 by john_rau.

Operation

Set a local register.

Format

```
setlocal
index
```

Forms

```
setlocal = 99 (0x63)
```

Stack

..., value => ...

Description

index is a u30 that must be an index of a local register. The register at index is set to value, and value is popped off the stack.

setlocal_n

This page last changed on Jun 23, 2010 by john_rau.

Operation

Set a local register.

Format

```
setlocal_<n>
```

Forms

```
setlocal_0 = 212 (0xd4)
setlocal_1 = 213 (0xd5)
setlocal_2 = 214 (0xd6)
setlocal_3 = 215 (0xd7)
```

Stack

..., value => ...

Description

<n> is an index of a local register. The register at that index is set to value, and value is popped off the stack.

setproperty

This page last changed on Jun 23, 2010 by john_rau.

Operation

Set a property.

Format

```
setproperty
index
```

Forms

`setproperty = 97 (0x61)`

Stack

..., obj, [ns], [name], value => ...

Description

value is the value that the property will be set to. value is popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

The property with the name specified by the multiname will be resolved in obj, and will be set to value. If the property is not found in obj, and obj is dynamic then the property will be created and set to value. See "Resolving multinames" on page 10.

Runtime exceptions

A TypeError is thrown if obj is `null` or `undefined`.

A ReferenceError is thrown if the property is const, or if the property is unresolved and obj is not dynamic.

setslot

This page last changed on Jun 23, 2010 by john_rau.

Operation

Set the value of a slot.

Format

```
setslot
slotindex
```

Forms

`setslot = 109 (0x6d)`

Stack

..., obj, value => ...

Description

slotindex is a u30 that must be an index of a slot on obj. slotindex must be greater than 0 and less than or equal to the total number of slots obj has.

This will set the value stored in the slot at slotindex on obj to value. value is first coerced to the type of the slot at slotindex.

Runtime exceptions

A TypeError is thrown if obj is `null` or `undefined`.

setsuper

Operation

Sets a property in a base class.

Format

```
setsuper
index
```

Forms

`setsuper = 5 (0x05)`

Stack

..., obj, [ns], [name], value => ...

Description

value is the value that the property will be set to. value is popped off the stack and saved.

index is a u30 that must be an index into the multiname constant pool. If the multiname at that index is a runtime multiname the name and/or namespace will also appear on the stack so that the multiname can be constructed correctly at runtime.

Once the multiname is constructed the base class of obj is determined and the multiname is resolved in the declared traits of the base class. The property is then set to value. See "Resolving multinames" on page 10.

Runtime exceptions

A TypeError is thrown if obj is `null` or `undefined`.

A ReferenceError is thrown if the property is unresolved, or if the property is read-only.

sf32

This page last changed on Jun 23, 2010 by john_rau.

Operation

Store single-precision floating point indirect.

Format

```
sf32
```

Forms

```
sf32 = 61 (0x3d)
```

Stack

...,value, offset => ...,

Description

Pop value and offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Store value as a single-precision (32-bit) floating point number at that offset in the ByteArray.

sf64

This page last changed on Jun 23, 2010 by john_rau.

Operation

Store double-precision floating point indirect.

Format

```
sf32
```

Forms

```
sf32 = 61 (0x3d)
```

Stack

...,value, offset => ...,

Description

Pop value and offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Store value as a double-precision (64-bit) floating point number at that offset in the ByteArray.

si16

Operation

Store short indirect.

Format

```
si16
```

Forms

```
si16 = 59 (0x3b)
```

Stack

...,value, offset => ...,

Description

Pop value and offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Store value as a 16-bit quantity at that offset in the ByteArray.

si32

Operation

Store int indirect.

Format

```
si32
```

Forms

```
si32 = 60 (0x3c)
```

Stack

...,value, offset => ...,

Description

Pop value and offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Store value as a 32-bit integer number at that offset in the ByteArray.

si8

Operation

Store byte indirect.

Format

```
si8
```

Forms

```
si8 = 58 (0x3a)
```

Stack

...,value, offset => ...,

Description

Pop value and offset off of the stack. The offset's value must lie between 0 and the size of the ByteArray which is the Domain's current `domainMemory` property. Store value as an 8-bit quantity at that offset in the ByteArray.

strictequals

This page last changed on Jun 23, 2010 by john_rau.

Operation

Compare two values strictly.

Format

```
strictequals
```

Forms

```
strictequals = 172 (0xac)
```

Stack

..., value1, value2 => ..., result

Description

Pop value1 and value2 off of the stack. Compare the two values using the Strict Equality Comparison Algorithm as described in ECMA-262 section 11.9.6. Push the resulting Boolean value onto the stack.

subtract

This page last changed on Jun 23, 2010 by john_rau.

Operation

subtract one value from another.

Format

```
subtract
```

Forms

```
subtract = 161 (0xa1)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack and convert value1 and value2 to Number to create value1_number and value2_number. Subtract value2_number from value1_number. Push the result onto the stack.

Notes

For more information, see ECMA-262 section 11.6 ("Additive Operators").

subtract_i

This page last changed on Jun 23, 2010 by john_rau.

Operation

Subtract an integer value from another integer value.

Format

```
subtract_i
```

Forms

```
subtract_i = 198 (0xc6)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack and convert value1 and value2 to int to create value1_int and value2_int. Subtract value2_int from value1_int. Push the result onto the stack.

swap

This page last changed on Jun 23, 2010 by john_rau.

Operation

Swap the top two operands on the stack

Format

```
swap
```

Forms

```
swap = 43(0x2b)
```

Stack

..., value1, value2 => ..., value2, value1

Description

Swap the top two values on the stack. Pop value2 and value1. Push value2, then push value1.

sxi_1

Operation

Sign-extend a single-bit value.

Format

```
sxi_1
```

Forms

```
sxi_1 = 80 (0x50)
```

Stack

..., value=> ...,result

Description

value popped off the stack and sign-extended by shifting left 31 bits and arithmetically shifting right 31 bits. The result is pushed onto the stack.

sxi_16

Operation

Sign-extend a 16-bit value.

Format

```
sxi_16
```

Forms

```
sxi_16 = 82 (0x52)
```

Stack

..., value=> ...,result

Description

value popped off the stack and sign-extended by shifting left 16 bits and arithmetically shifting right 16 bits. The result is pushed onto the stack.

sxi_8

Operation

Sign-extend an 8-bit value.

Format

```
sxi_8
```

Forms

```
sxi_8 = 81 (0x51)
```

Stack

..., value=> ...,result

Description

value popped off the stack and sign-extended by shifting left 24 bits and arithmetically shifting right 24 bits. The result is pushed onto the stack.

throw

This page last changed on Jun 23, 2010 by john_rau.

Operation

Throws an exception.

Format

```
throw
```

Forms

```
throw = 3 (0x03)
```

Stack

..., value => ...

Description

The top value of the stack is popped off the stack and then thrown. The thrown value can be of any type.

When a `throw` is executed, the current method's exception handler table is searched for an exception handler. An exception handler matches if its range of offsets includes the offset of this instruction, and if its type matches the type of the thrown object, or is a base class of the type thrown. The first handler that matches is the one used.

If a handler is found then the stack is cleared, the exception object is pushed onto the stack, and then execution resumes at the instruction offset specified by the handler.

If a handler is not found, then the method exits, and the exception is rethrown in the invoking method, at which point it is searched for an exception handler as described here.

typeof

This page last changed on Jun 23, 2010 by john_rau.

Operation

Get the type name of a value.

Format

```
typeof
```

Forms

```
typeof = 149 (0x95)
```

Stack

..., value => ..., typename

Description

Pop a value off of the stack. Determine its type name according to the type of value:

1. undefined = "`undefined`"
2. null = "`object`"
3. Boolean = "`Boolean`"
4. Number | int | uint = "`number`"
5. String = "`string`"
6. Function = "`function`"
7. XML | XMLList = "`xml`"
8. Object = "`object`"

Push typename onto the stack.

urshift

This page last changed on Jun 23, 2010 by john_rau.

Operation

Unsigned bitwise right shift.

Format

```
urshift
```

Forms

```
urshift = 167 (0xa7)
```

Stack

..., value1, value2 => ..., value3

Description

Pop value1 and value2 off of the stack, convert value1 to an int to create value1_int and convert value2 to a uint to create value2_uint. Right shift value1_int by the result of value2_uint & 0x1F (leaving only the 5 least significant bits of value2_uint), and push the result onto the stack. The right shift is unsigned and fills in missing bits with 0, resulting in an unsigned 32-bit integer. See ECMA-262 section 11.7.3

6. Hints for compiler writers

This page last changed on Jun 23, 2010 by john_rau.

The following techniques may be useful to compiler writers targeting the AVM2:

Reestablishing the scope stack following an exception

When an exception handler is entered, the scope stack established in the catching method has been cleared (see the description of `throw` in the previous chapter). This is not always desired; a `try..catch` statement inside a `with` statement requires the extended scope of the `with` to still be established. The way to work around the clearing of the scope stack is for the exception handler to reestablish the scope stack by pushing the correct set of objects onto it before executing the body of the handler. That is, in this code:

```
with (x) {
try {
...
}
catch (e) {
return y
}
}
```

the body of the `catch` clause needs to look something like this, assuming `x` is in local 0:

```
getlocal_0
pushwith
...                     ; set up catch scope
findproperty "y"
getproperty "y"
returnvalue
```

In-line subroutines for handling "finally"

A `finally` clause on a `try` block must be visited when control flows out of the `try` block (or out of the associated `catch` clause). It is possible to expand the body of the `finally` clause in-line everywhere it needs to be visited, but this tends to greatly increase the amount of compiled code in the program. A more reasonable approach is to expand the `finally` clause once and visit it by means of the `jump` instruction. To have this "subroutine" return to the point from which it was "called", the "caller" stores a value identifying itself in a local variable. The `finally` clause returns to this location by means of a `lookupswitch` instruction:

```
pushshort 0
setlocal_0
jump    L1
R1      ...                     ; Code following subroutine call
...
pushshort 1
setlocal_0
jump    L1
R2      ...                     ; Code following subroutine call
...
L1                              ; Subroutine.  Return "address" in local 0
...                     ; Body of subroutine
getlocal_0
lookupswitch
0 -> R1
1 -> R2
```

(This only highlights the fact that `lookupswitch` is a computed goto.)

Creating an environment for nested functions

ActionScript functions that are passed around as values close over their environment, including the environment's local variables, when they are created. Since the local registers of an activation are not captured when the `newfunction` instruction is executed, the environment for non-leaf functions must be stored in activation objects that can be captured properly. The `newactivation` instruction creates such an activation. Given the source code:

```
function f(x) {
return function () { return x }
}
```

a suitable translation is along the lines of

```
newactivation            ; create a new activation record
dup                     ;   and save a copy
pushscope               ;      and extend the current scope
getlocal_o              ; get x parameter
setproperty "x" ; store x in the activation
newfunction <inner>     ; create a new function
returnvalue
```

where `<inner>` is just a reference to the method body for the nested function. Note that `newfunction` captures the contents of the scope stack, but not the local registers or the operand stack.

Optimizing a method dispatch

The instructions `callmethod` and `callstatic` can be used to optimize method dispatch. They require that the method properties of the receiver object be laid out in a particular order; the compiler provides explicit non-zero offsets for the methods in the `trait_method` structure in the `abcFile`. The call instructions then call directly through the method table offsets, avoiding the name lookup. The verifier needs to be able to determine that the receiver object is of a type that has a method table that has a method at that offset. Usually this means that the receiver object can only be read by the bytecode from type-annotated locations, or that the call instruction must be preceded by a `coerce` instruction.

Resources

This page last changed on Jun 07, 2010 by john_rau.

ActionScript Virtual Machine 2

PDF Version

RSS Feed

Resources

- Adobe.com
- Adobe Feeds
- Adobe Labs
- Document Services
- Kuler

empty

This page last changed on Jun 23, 2010 by john_rau.

empty2

This page last changed on Jun 23, 2010 by john_rau.

empty3

This page last changed on Jun 23, 2010 by john_rau.

empty4

This page last changed on Jun 23, 2010 by john_rau.