NGWS runtime

IL Instruction Set Specification

Version 1.9 Final

Last updated: 8 June 2000

This is preliminary documentation and subject to change

Table of Contents

1 Introduction to the Runtime IL Instruction Set

This specification is a detailed description of the NGWS runtime Intermediate Language (IL) instruction set. The runtime Execution Engine Architecture Specification, Architecture.doc, describes the architecture of the runtime Execution Engine (EE) and provides an overview of a large number of issues relating to the IL instruction set. That overview is essential to an understanding of the instruction set as described here.

Each instruction description describes a set of related EE machine instructions. Each instruction definition consist of five parts:

- A table describing the binary format, assembly language notation and description of each variant of the instruction. See the Instruction Variant Table section.
- A stack transition diagram that describes the state of the evaluation stack before and after the instruction is executed. See the Stack Transition Diagram section.
- An English description of the instruction. See the English Description section.
- A list of exceptions that might be thrown by the instruction. See the EE Architecture Specification for more details.
- A section describing the verifiability conditions associated with the instruction. See the Verifiability section.

In addition, operations that have a numeric operand also specify an operand type table that describes how they operate based on the type of the operand. See the Operand Type Table section.

1.1 Data Types

While the Virtual Object System defines a rich type system and the Common Language Specification specifies a subset that can be used for language interoperability, the Execution Engine itself deals with a much simpler set of types. These types, collectively known as the "basic EE types," are:

- A subset of the full numeric types (I4, I8, I, and F)
- Object references (O) but without distinction between the type of object referenced
- Pointer types (U, *, and &) without distinction as to the type pointed to

1.1.1 Numeric Data Types

The EE only tracks the numeric types I4 (4 byte signed integers), I8 (8 byte signed integers), I (native size integers), and F (native size floating point numbers). The IL instruction set, however, allows additional data types to be implemented:

- **Short integers**. The model is that the evaluation stack only holds 4 or 8 byte integers, but other locations (arguments, local variables, statics, array elements, fields) may hold 1 or 2 byte integers. Loading from these locations onto the stack either zero-extends (**ldind.u***, **ldelem.u***, etc.) or sign-extends (**ldind.i***, **ldelem.i***, etc.) to a 4 byte value. Storing to integers (**stind.u1**, **stelem.i2**, etc.) truncates. Use the **conv.ovf.*** instructions to detect when this truncation results in a value that doesn't correctly represent the original value.

Convert instructions that yield short integer values actually leave an I4 (32-bit) value on the stack, but it is guaranteed that only the low bits have meaning (i.e. the more significant bits are all zero for the unsigned conversions or a sign extension for the signed conversions). To correctly simulate the full set of short integer operations a conversion to the short form is required before the **div**, **rem, shr**, comparison and conditional branch instructions.

In addition to the explicit conversion instructions there are four cases where the EE handles short integers in a special way:

1. Assignment to a local (**stloc**) or argument (**starg**) whose type is declared to be a short integer type automatically truncates to the size specified for the local or argument.
2. Loading from a local (**ldloc**) or argument (**ldarg**) whose type is declared to be a short signed integer type automatically sign extends.
3. Calling a procedure with an argument that is a short integer type is equivalent to assignment to the argument value, so it truncates.
4. Returning a value from a method whose return type is a short integer can be thought of as storing into a short integer within the called procedure (i.e. the EE automatically truncates) and loading from a short integer within the calling procedure (i.e. the EE automatically zero- or sign-extends).

In the last two cases it is up to the native calling convention to determine whether values are actually truncated or extended, as well as whether this is done in the called procedure or the calling procedure. The IL instruction sequence is unaffected and it is as though the IL sequence included an appropriate **conv** instruction.

- **4 byte integers**. The shortest value actually stored on the stack is a 4-byte integer. These can be converted to 8-byte integers or native-size integers using **conv.*** instructions. Native-size integers can be converted to 4-byte integers, but doing so is not portable across architectures. The **conv.i4** and **conv.u4** can be used for this conversion if loss of precision is desirable; the **conv.ovf.i4** and **conv.ovf.u4** instructions can be used to detect the loss of information. Arithmetic operations allow 4-byte integers to be combined with native size integers, resulting in native size integers. 4-byte integers may not be directly combined with 8-byte integers (they must be converted to 8-byte integers first).
- **Native size integers**. Native size integers can be combined with 4-byte integers using any of the normal arithmetic instructions, and the result will be a native-size integer. Native size integers must be explicitly converted to 8-byte integers before they can be combined with 8-byte integers.
- **8 byte integers**. Supporting 8 byte integers on 32 bit hardware is expensive, whereas 32 bit arithmetic is available and efficient on current 64 bit hardware. For this reason, numeric instructions allow I4 and I data types to be intermixed (yielding the largest type used as input), but these types *cannot* be combined with I8s. Instead, an I or I4 must be explicitly converted to I8 before it can be combined with an I8.
- **Unsigned integers**. Special instructions are used to interpret integers on the stack as though they were unsigned, rather than tagging the stack locations as being unsigned.
- **Floating point numbers**. Storage locations for floating point numbers (statics, array elements, and fields of classes) are of fixed size. The supported storage sizes are **R4** (4 byte real numbers in IEEE754 single precision format), **R8** (8 byte real numbers in IEEE754 double precision format), and **RPrecise** (a fixed size for any given

architecture, at least 64 bits wide, and as precise as can be efficiently supported on that architecture).

Everywhere else (on the evaluation stack, as arguments, as return types, and as local variables) floating point numbers are represented using the internal **F** type. This type can be thought of as starting at the size of value loaded from storage and then expanding as needed. This design allows the EE to choose a platform-specific high-performance representation for floating point numbers until they are placed in storage locations. For example, it may be able to leave floating point variables in hardware registers that provide more precision than a user has requested. At the same time, IL generators can force operations to respect language-specific rules for representations through the use of conversion instructions.

When a value of type **F** is put in a storage location it is automatically coerced to the required size, which may involve a loss of precision or the creation of an out-of-range marker (a **NaN**). To detect values that cannot be converted to a particular storage type, use a conversion instruction (**conv.r4**, **conv.r8**, **conv.r4result**, **conv.r8result**, or **conv.rprecise**) and then check for a non-finite value using **ckfinite**. To detect underflow when converting to a particular storage type, a comparison to zero is required before and after the conversion.

1.1.2 Object References

Object references (type O) are completely opaque. There are no arithmetic instructions that allow object references as operands, and the only comparison operations permitted are equality (and inequality) between two object references. There are no conversion operations defined on object references. Object references are created by certain IL object instructions (notably **newobj** and **newarr**). Object references can be passed as arguments, stored as local variables, returned as values, and stored in arrays and as fields of objects.

1.1.3 Runtime Pointer Types

There are three kinds of pointers: unmanaged pointer, managed pointers, and transient pointers. For pointers into the same array or object (see the EE Architecture Specification_), the following arithmetic operations are defined:

- Adding an integer to a pointer, where the integer is interpreted as a number of bytes, results in a pointer of the same kind.
- Subtracting an integer (number of bytes) from a pointer results in a pointer of the same kind. Note that subtracting a pointer from an integer is not permitted.
- Two pointers, regardless of kind, can be subtracted from one another, producing an integer that specifies the number of bytes between the addresses they reference.

None of these operations is allowed in verifiable code.

It is important to understand the impact on the garbage collector of using arithmetic on the different kinds of pointers. Since unmanaged pointers never reference memory that is controlled by the garbage collector, performing arithmetic on them can endanger the memory safety of the system (hence it is not verifiable) but since they are not reported to the garbage collector there is no impact on its operation. Similarly, transient pointers are not reported to the garbage collector and arithmetic can be performed without impact on garbage collection.

Managed pointers, however, are reported to the garbage collector. As part of garbage collection both the contents of the location to which they point *and* the pointer itself can be modified. The garbage collector will ignore managed pointers if they point into memory that is not under its control (the evaluation stack, the call stack, static memory, or memory under the control of another allocator). If, however, a managed pointer refers to memory controlled by the garbage collector it must point to either a field of an object, an element of an array, or the address of the element just past the end of an array. If address arithmetic is used to create a managed pointer that refers to any other location (an object header or a gap in the allocated memory) the garbage collector's operation is unspecified.

1.1.3.1 Unmanaged Pointers

Unmanaged pointers are the traditional pointers used in languages like C and C++. There are no restrictions on their use, although for the most part they result in code that cannot be verified. While it is perfectly legal to mark locations that contain unmanaged pointers as though they were unsigned integers (and this is, in fact, how they are treated by the EE), it is often better to mark them as unmanaged pointers to a specific type of data. This is done by using **ELEMENT_TYPE_PTR** in a signature for a return value, local variable or an argument (see the Metadata Structures Specification) or by using a pointer type for a field or array element.

Unmanaged pointers are not reported to the garbage collector and can be used in any way that an integer can be used.

- It is best to think of unmanaged pointers as unsigned (i.e. use **conv.ovf.u** rather than **conv.ovf.i**, etc.).
- Verifiable code cannot use unmanaged pointers to reference memory (i.e. it treats them as integers, not pointers).
- Unverified code can pass an unmanaged pointer to a method that expects a managed pointer. This is safe only if one of the following is true:
    1. The unmanaged pointer refers to memory that is not in memory managed by the garbage collector
    2. The unmanaged pointer refers to a field within an object
    3. The unmanaged pointer refers to an element within an array
    4. The unmanaged pointer refers to the location where the element following the last element in an array would be located

1.1.3.2 Managed Pointers (type &)

Managed pointers (**&**) may point to a field of an object, a field of a value type, an element of an array, or the address where an element just past the end of an array would be stored (for pointer indexes into managed arrays). Managed pointers cannot be **null**, and they must be reported to the garbage collector, even if they do not point to managed memory.

Managed pointers are specified by using **ELEMENT_TYPE_BYREF** in a signature for a return value, local variable or an argument or by using a by-ref type for a field or array element. See the Metadata Structures Specification for details.

- Managed pointers can be passed as arguments and stored in local variables.
- If you pass a parameter by reference, the corresponding argument is a managed pointer.
- Managed pointers cannot be stored in static variables, array elements, or fields of objects or value types.
- Managed pointers are *not* interchangeable with object references.
- A managed pointer cannot point to another managed pointer, but it can point to an object reference or a value type.
- Managed pointers that do not point to managed memory can be converted (using **conv.u** or **conv.ovf.u**) into unmanaged pointers, but this is not verifiable.
- Unverified code that erroneously converts a managed pointer into an unmanaged pointer can seriously compromise the integrity of the EE. This conversion is only safe if one of the following is known to be true:
  1. the managed pointer does not point into the garbage collector's memory area
  2. the memory referred to has been pinned for the entire time that the unmanaged pointer is in use
  3. a garbage collection cannot occur while the unmanaged pointer is in use

1.1.3.3Transient Pointers (type *)

Transient pointers (*****) are intermediate between managed and unmanaged pointers. They are created within the EE by certain IL instructions, but users cannot declare locations of this type. When a transient pointer is passed as an argument, returned as a value, or stored into a user-visible location it is converted either to a managed pointer or an unmanaged pointer depending on the type specified for the destination.

- The IL instructions that create transient pointers (**ldloca, ldarga, ldsflda** when the type of the field is *not* an object) are guaranteed to produce pointers to data that is *not* in managed memory.
- Transient pointers need *not* be reported to the garbage collector, and they are automatically converted to managed or unmanaged pointers when necessary (on method call or when stored into a local or argument that requires a managed pointer).
- Transient pointers can exist only on the evaluation stack within a single method.
- The verifier treats transient pointers as managed pointers.

1.2Instruction Variant Table

Each variant of an instruction is described in this table. The format column of the table describes the reference opcode assigned to the instruction variant, along with any arguments that follow the instruction in the instruction stream. A typical instruction format entry might look like

FE 0A <**U2**>

Boldface numbers represent literal bytes in the instruction stream. In the example above the instruction is encoded by the byte **FE** followed by the byte **0A.** Italicized type names represent numbers that should follow in the instruction stream. In the example above a 2-

byte quantity that is to be treated as an unsigned integer directly follows the **FE 0A** opcode.

Any of the fixed size primitive types (I1, U1, I2, U2, I4, U4, I8, U8, R4, and R8) can appear in opcode format descriptions. These types define the number of bytes for the argument and how it should be interpreted (signed, unsigned or floating point). In addition, a metadata token can appear, indicated as <**T**>. Tokens are encoded as a 4-byte unsigned integer. All argument numbers are encoded least significant byte first (little endian). Bytes for instruction opcodes and arguments are packed as tightly as possible (no alignment padding is done).

The assembly format column defines an assembly code mnemonic (see the Assembler Programmers' Reference) for the instruction variant. For those instructions that have instruction stream arguments, this column also assigns names to each of the arguments to the instruction. For each instruction argument, there is a name in the assembly format. These names are used later in the instruction description.

1.2.1 Opcode Encodings

The IL opcode space is encoded using one, two, and four byte opcodes. One byte encodings range from 0x00 through 0xEF and are all reserved, even if not currently in use. Two byte encodings start with 0xF0 through 0xFE. Of these, the set beginning with 0xFE are reserved for future expansion by Microsoft and the other sets are unspecified and must not be used. Four byte encodings begin with 0xFF and must not be used.

The currently defined encodings are specified in Table 1

0x00	nop
0x01	break
0x02	ldarg.0
0x03	ldarg.1
0x04	ldarg.2
0x05	ldarg.3
0x06	ldloc.0
0x07	ldloc.1
0x08	ldloc.2
0x09	ldloc.3
0x0a	stloc.0
0x0b	stloc.1
0x0c	stloc.2
0x0d	stloc.3
0x0e	ldarg.s
0x0f	ldarga.s
0x10	starg.s
0x11	ldloc.s
0x12	ldloca.s
0x13	stloc.s
0x14	ldnull
0x15	ldc.i4.m1
0x16	ldc.i4.0
0x17	ldc.i4.1
0x18	ldc.i4.2
0x19	ldc.i4.3
0x1a	ldc.i4.4
0x1b	ldc.i4.5
0x1c	ldc.i4.6
0x1d	ldc.i4.7
0x1e	ldc.i4.8
0x1f	ldc.i4.s
0x20	ldc.i4
0x21	ldc.i8
0x22	ldc.r4
0x23	ldc.r8

0x25	dup

0x26	pop

0x27	jmp
0x28	call
0x29	calli
0x2a	ret
0x2b	br.s
0x2c	brfalse.s
0x2d	brtrue.s
0x2e	beq.s
0x2f	bge.s
0x30	bgt.s
0x31	ble.s
0x32	blt.s
0x33	bne.un.s
0x34	bge.un.s
0x35	bgt.un.s
0x36	ble.un.s
0x37	blt.un.s
0x38	br
0x39	brfalse
0x3a	brtrue
0x3b	beq
0x3c	bge
0x3d	bgt
0x3e	ble
0x3f	blt
0x40	bne.un
0x41	bge.un
0x42	bgt.un
0x43	ble.un
0x44	blt.un
0x45	switch
0x46	ldind.i1
0x47	ldind.u1
0x48	ldind.i2

0x49	ldind.u2
0x4a	ldind.i4
0x4c	ldind.i8
0x4d	ldind.i
0x4e	ldind.r4
0x4f	ldind.r8
0x50	ldind.ref
0x51	stind.ref
0x52	stind.i1
0x53	stind.i2
0x54	stind.i4
0x55	stind.i8
0x56	stind.r4
0x57	stind.r8
0x58	add
0x59	sub
0x5a	mul
0x5b	div
0x5c	div.un
0x5d	rem
0x5e	rem.un
0x5f	and
0x60	or
0x61	xor
0x62	shl
0x63	shr
0x64	shr.un
0x65	neg
0x66	not
0x67	conv.i1
0x68	conv.i2
0x69	conv.i4
0x6a	conv.i8
0x6b	conv.r4
0x6c	conv.r8

0x6d	conv.u4
0x6e	conv.u8
0x6f	callvirt
0x70	cpobj
0x71	ldobj
0x72	ldstr
0x73	newobj
0x74	castclass
0x75	isinst
0x76	conv.r.un
0x77	ann.data.s
0x78	box
0x79	unbox
0x7a	throw
0x7b	ldfld
0x7c	ldflda
0x7d	stfld
0x7e	ldsfld
0x7f	ldsflda
0x80	stsfld
0x81	stobj
0x82	conv.ovf.i1.un
0x83	conv.ovf.i2.un
0x84	conv.ovf.i4.un
0x85	conv.ovf.i8.un
0x86	conv.ovf.u1.un
0x87	conv.ovf.u2.un
0x88	conv.ovf.u4.un
0x89	conv.ovf.u8.un
0x8a	conv.ovf.i.un
0x8b	conv.ovf.u.un
0x8d	newarr
0x8e	ldlen
0x8f	ldelema
0x90	ldelem.i1
0x91	ldelem.u1
0x92	ldelem.i2
0x93	ldelem.u2
0x94	ldelem.i4
0x96	ldelem.i8
0x97	ldelem.i
0x98	ldelem.r4
0x99	ldelem.r8
0x9a	ldelem.ref
0x9b	stelem.i
0x9c	stelem.i1
0x9d	stelem.i2
0x9e	stelem.i4
0x9f	stelem.i8
0xa0	stelem.r4
0xa1	stelem.r8
0xa2	stelem.ref
0xb3	conv.ovf.i1
0xb4	conv.ovf.u1
0xb5	conv.ovf.i2
0xb6	conv.ovf.u2
0xb7	conv.ovf.i4
0xb8	conv.ovf.u4
0xb9	conv.ovf.i8
0xba	conv.ovf.u8
0xc2	refanyval
0xc3	ckfinite
0xc6	mkrefany
0xc7	ann.call
0xc8	ann.catch
0xc9	ann.dead
0xca	ann.hoisted
0xcb	ann.hoisted_call
0xcc	ann.lab

0xcd	ann.def
0xce	ann.ref.s
0xcf	ann.phi
0xd0	ldtoken
0xd1	conv.u2
0xd2	conv.u1
0xd3	conv.i
0xd4	conv.ovf.i
0xd5	conv.ovf.u
0xd6	add.ovf
0xd7	add.ovf.un
0xd8	mul.ovf
0xd9	mul.ovf.un
0xda	sub.ovf
0xdb	sub.ovf.un
0xdc	endfinally
0xdd	leave
0xde	leave.s
0xdf	stind.i
0xe0	conv.u
0xfe 0x00	arglist
0xfe 0x01	ceq
0xfe 0x02	cgt
0xfe 0x03	cgt.un
0xfe 0x04	clt
0xfe 0x05	clt.un
0xfe 0x06	ldftn
0xfe 0x07	ldvirtftn
0xfe 0x08	jmpi
0xfe 0x09	ldarg
0xfe 0x0a	ldarga
0xfe 0x0b	starg
0xfe 0x0c	ldloc
0xfe 0x0d	ldloca
0xfe 0x0e	stloc
0xfe 0x0f	localloc
0xfe 0x11	endfilter
0xfe 0x12	unaligned.
0xfe 0x13	volatile.
0xfe 0x14	tail.
0xfe 0x15	initobj
0xfe 0x16	ann.live
0xfe 0x17	cpblk
0xfe 0x18	initblk
0xfe 0x19	ann.ref
0xfe 0x1a	rethrow
0xfe 0x1c	sizeof
0xfe 0x1d	refanytype
0xfe 0x22	ann.data
0xfe 0x23	ann.arg

Table 1: Opcode Encodings

1.3 Stack Transition Diagram

The stack transition diagram displays the state of the evaluation stack before and after the instruction is executed. Below is a typical stack transition diagram.

```
…, value1, value2


…, result
```

This diagram indicates that the stack must have at least two elements on it, and in the definition the topmost value ("top of stack" or "most recently pushed") will be called *value2* and the value underneath (pushed prior to *value2*) will be called *value1*. (In diagrams like this, the stack grows to the right, along the page). The instruction removes these values from the stack and replaces them by another value, called *result* in the description.

1.4 English Description

The English description describes any details about the instructions that are not immediately apparent once the format and stack transition have been described.

1.5 Verifiability

Memory safety is a property that ensures programs running in the same address space are correctly isolated from one another (see the EE Architecture Specification_). Thus, it is desirable to test whether programs are memory safe prior to running them. Unfortunately, it is provably impossible to do this with 100% accuracy. Instead, the runtime can test a stronger restriction, called *verifiability*. Every program that is verified is memory safe, but some programs that are not verifiable are still memory safe.

It is perfectly acceptable to generate IL code that is not verifiable, but which is known to be memory safe by the compiler writer. Several important uses of IL instructions are not verifiable, such as the pointer arithmetic versions of **add** that are required for the faithful and efficient compilation of C programs. For non-verifiable code, memory safety is the responsibility of the compiler writer.

IL contains a *verifiable subset*. The Verifiability description gives details of the conditions under which a use of an instruction falls within the verifiable subset of IL. The verifier tracks the types of values in much finer detail than is required for the basic functioning of the EE, because it is checking that an IL code sequence respects not only the basic rules of the EE with respect to the safety of garbage collection, but also the typing rules of the Virtual Object System. This helps to guarantee the sound operation of the entire EE. The descriptions therefore make the distinction between "verification types" and "EE types", the latter already having been described in Section 1.1.

The verifiability section of each operation description specifies requirements both for correct IL generation and for verification. Correct IL generation always requires guaranteeing that the top items on the stack correspond to the types shown in the stack transition diagram. The verifiability section specifies only requirements for correct IL generation that not captured in that diagram. The verifier tests both the requirements for correct IL generation and the specific verification conditions that are described with the instruction. The operation of IL sequences that do not meet the IL correctness requirements is unspecified. The operation of IL sequences that meet the correctness requirements but is not verifiable may violate typesafety and hence violate security or memory access constraints.

1.6 Operand Type Table

Many IL operations take numeric operands on the stack. These fall into several different categories, depending on how they deal with the types of the operands. The following operand tables summarize the legal operand types and the resulting type. Notice that the type referred to here is the type as tracked by the Execution Engine rather than the more detailed types used by tools such as the IL verifier. The types tracked by the EE are: I4, I8, I, F, O, &, and *.

Table 2: Binary Numeric Operations

A **op** B (used for **add**, **div**, **mul**, **rem**, and **sub**, applies to all instructions unless specific instructions are specified in the table). The shaded uses are not verifiable, while items marked "-" indicate incorrectly formed IL sequences.

B's type A's type	I4	I8	I	F	&	O	*
I4	I4	-	I	-	& (add)	-	* (add)
I8	-	I8	-	-	-	-	-
I	I	-	I	-	& (add)	-	* (add)
F	-	-	-	F	-	-	-
&	& (add, sub)	-	& (add, sub)	-	I (sub)	-	I (sub)
O	-	-	-	-	-	-	-
*	* (add, sub)	-	* (add, sub)	-	I (sub)	-	I (sub)

Table 3: Unary Numeric Operations

Used for the **neg** instruction. All these uses of this instruction are verifiable.

Operand Type	I4	I8	I	F	&	O	*
Result Type	I4	I8	I	F	-	-	-

Table 4: Binary Comparison or Branch Operations

These return a boolean value or branch based on the top two values on the stack. Used for **beq, bge, bge.un, bgt, bgt.un, ble, ble.un, blt, blt.un, bne, bne.un, ceq, cgt, cgt.un, clt, clt.un**. Items marked "✓" indicate that all instructions are valid. Items marked "-" indicate invalid IL sequences. If only a subset of instructions are permitted, the valid instructions are shown in the corresponding cell.

	I4	I8	I	F	&	O	*
I4	✓	-	✓	-	-	-	-
I8	-	✓	-	-	-	-	-
I	✓	-	✓	-	beq[.s], bne.un[.s], ceq	-	beq[.s], bne.un[.s], ceq
F	-	-	-	✓	-	-	-

&	-	-	**beq[.s], bne.un[.s], ceq**	-	**✓ (Note)**	-	**✓ (Note)**
O	-	-	-	-	-	**beq[.s], bne.un[.s], ceq**	-
*****	-	-	**beq[.s], bne.un[.s], ceq**	-	**✓ (Note)**	-	**✓ (Note)**

Note: Except for beq, bne.un (or short versions) or ceq these combinations only make sense if both operands are known to be pointers to elements of the same array.

Table 5: Integer Operations

These operate only on integer types. Used for **and, div.un, not, or, rem.un, shl, shr, xor**. The **div.un** and **rem.un** instructions treat their arguments as unsigned integers and produce the bit pattern corresponding to the unsigned result. As described in the EE Specification, however, the EE makes no distinction between signed and unsigned integers on the stack. The **not** instruction is unary and returns the same type as the input. The **shl** and **shr** instructions return the same type as their first operand and their second operand must be of type U. All items marked "-" indicate incorrectly formed IL sequences, while the others are verifiable.

	I4	**I8**	**I**	**F**	**&**	**O**	*****
I4	I4	-	I	-	-	-	-
I8	-	I8	-	-	-	-	-
I	I	-	I	-	-	-	-
F	-	-	-	-	-	-	-
&	-	-	-	-	-	-	-
O	-	-	-	-	-	-	-
*****	-	-	-	-	-	-	-

Table 6: Overflow Arithmetic Operations

These operations generate an exception if the result cannot be represented in the target data type. Used for **add.ovf, add.ovf.un, mul.ovf, mul.ovf.un, sub.ovf, sub.ovf.un** The shaded uses are not verifiable, while items marked "-" indicate incorrectly formed IL sequences.

	I4	**I8**	**I**	**F**	**&**	**O**	*****
I4	**I4**	**-**	**I**	**-**	**& add.ovf.un**	**-**	*** add.ovf.un**
I8	**-**	**I8**	**-**	**-**	**-**	**-**	**-**
I	**I**	**-**	**I**	**-**	**& add.ovf.**	**-**	*** add.ovf.**

					un		un
F	-	-	-	-	-	-	-
&	& add.ovf.un, sub.ovf.un	-	& add.ovf.un, sub.ovf.un	-	I sub.ovf.un	-	I sub.ovf.un
O	-	-	-	-	-	-	-
*	* add.ovf.un, sub.ovf.un	-	* add.ovf.un, sub.ovf.un	-	I sub.ovf.un	-	I sub.ovf.un

Table 7: Conversion Operations

These operations convert from one numeric type to another. The result type is guaranteed to be representable as the data type specified as part of the operation (i.e. the **conv.u2** instruction returns a value that can be stored in a **U2**). The stack, however, can only store values that are a minimum of 4 bytes wide. Used for the **conv.<to type>, conv.ovf.<to type>**, and **conv.ovf.<to type>.un** instructions. The shaded uses are not verifiable, while items marked "-" indicate incorrectly formed IL sequences.

Output Operand	I1/U1 I2/U2	I4/U4	I8	U8	I
I4	Truncate[1]	No-op	Sign extend	Zero extend	Sign extend
I8	Truncate[1]	Truncate[1]	No-op	No-op	Truncate[1]
I	Truncate[1]	Truncate[1]	Sign extend	Zero extend	No-op
F	Trunc to 0[2]	Trunc to 0[2]	Trunc to 0[2]	Trunc to 0[2]	Trunc to 0[2]
&	-	-	-	Stop GC Tracking	-
O	-	-	-	-	-
*	-	-	-	Zero extend	-

Output Operand	U	All R Types	&
I4	Zero extend	To Float	-
I8	Truncate[1]	To Float	-
I	No-op	To Float	-
F	Trunc to 0[2]	Change Precision[3]	-
&	Stop GC Tracking	-	No-op
O	-	-	-

*	**No-op**	**-**	**Start GC Tracking**

Note 1: "Truncate" means that the number is truncated (i.e. the higher-order bits are set to zero) to the desired size. If the destination type is signed, the most-significant bit of the truncated value is then sign-extended to fill the full output size. Thus, converting 257 (0x101) to I1 or U1 yields 1, but truncating 129 (0x81) to U1 yields 129 (0x81) while truncating it to I1 yields –126 (0xF...F81)

Note 2: "Trunc to 0" means that the floating point number will be converted to an integer by truncation toward zero. Thus 1.1 is converted to 1 and –1.1 is converted to –1.

Note 3: Converts from the current precision available on the evaluation stack to the precision specified by the instruction. If the stack has more precision than the output size the conversion is performed using the IEEE 754 "round to nearest" mode to compute the low order bit of the result.

1.7 Signature Matching

While the EE deals only with 7 types (I4, I, I8, F, O, &, and *) the metadata supplies a much richer model for parameters of methods. The verifier is responsible for ensuring the detailed type matching expected for memory safety. The shaded uses are not verifiable, while the items marked "No" are incorrect IL sequences. The EE uses the following rules when passing data from one method to another:

Stack Parameter	**I4**	**I**	**I8**	**F**	**&**	**O**	*****
I1	Note 1	Note 1	No	No	No	No	No
U1	Note 1	Note 1	No	No	No	No	No
I2	Note 1	Note 1	No	No	No	No	No
U2	Note 1	Note 1	No	No	No	No	No
I4	OK	Note 1	No	No	No	No	No
U4	As is	Note 2	No	No	No	No	No
I8	Sign extend	Sign extend	OK	No	No	No	No
U8	Zero extend	Zero extend	OK	No	No	No	No
I	Sign extend	OK	No	No	No	No	No
U	Zero extend	Zero extend	No	No	**Note 3**	No	**No GC tracking**
R4	No	No	No	Round	No	No	No
R8	No	No	No	Round	No	No	No
R	No	No	No	Round	No	No	No

Class	No	No	No	No	No	OK	No
Value Type (Note 4)	Note 5	Note 5	Note 5	Note 5	No	No	No
By-Ref (&)	No	**Start GC tracking**	No	No	OK	No	Start GC tracking
Ref Any (Note 6)	No	No	No	No	No	No	No

1. The EE provides an implicit **conv.*** instruction to generate the correct parameter type.
2. On a 32-bit machine passing an **I** argument to a **U4** parameter involves no conversion. On a 64-bit machine it is treated as described in note 1.
3. See Section 1.1.3.2. This conversion is *not* provided automatically by the EE.
4. The EE's stack can contain a value type. These may only be passed if the particular value type on the stack exactly matches the class required by the corresponding parameter.
5. Passing a primitive type to a parameter that is required to be a value type is not allowed
6. There are special instructions to construct and pass a **Ref Any**.

2 Base Instructions

These instructions form a “Turing Complete” set of basic operations. They are independent of the object model that may be employed. Operations that are specifically related to the Virtual Object System’s object model are contained in the Object Model Instructions section. Annotations, which are used with OptIL but can be ignored by most IL processors, are described in the Annotations section.

add - add numeric values

Format	Assembly Format	Description
58	add	Add two values, returning a new value

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The **add** instruction adds *value2* to *value1* and pushes the result on the stack. Overflow is not detected for integral operations (but see **add.ovf**); floating point overflow returns **+inf** or **-inf**.

The acceptable operand types and their corresponding result data type is encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 2: Binary Numeric Operations**.**

add.ovf.<signed> - add integer values with overflow check

Format	Assembly Format	Description
D6	add.ovf	Add signed integer values with overflow check.
D7	add.ovf.un	Add unsigned integer values with overflow check.

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The **add.ovf** instruction adds *value1* and *value2* and pushes the result on the stack. The acceptable operand types and their corresponding result data type is encapsulated in Table 6: Overflow Arithmetic Operations.

Exceptions:

OverflowException is thrown if the result can not be represented in the result type.

Verifiability:

See Table 6: Overflow Arithmetic Operations.

and - bitwise AND

Format	Instruction	Description
5F	and	Bitwise AND of two integral values, returns an integral value

Stack Transition:

```
…, value1, value2


…, result
```

Description:

The **and** instruction computes the bitwise AND of the top two values on the stack and pushes the result on the stack. The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

arglist - get argument list

Format	Assembly Format	Description
FE 00	arglist	return argument list handle for the current method

Stack Transition:

…

…, argListHandle

Description:

The **arglist** instruction returns an opaque handle (an unmanaged pointer, type **I**) representing the argument list of the current method. This handle is valid only during the lifetime of the current method. The handle can, however, be passed to other methods as long as the current method is on the thread of control. The **arglist** instruction may only be executed within a method that takes a variable number of arguments.

Rationale: *This instruction is needed to implement the C 'va_*' macros used to implement procedures like 'printf'. It is intended for use with the class library implementation of **System.ArgIterator**.*

Exceptions:

None.

Verifiability:

It is incorrect IL generation to emit this instruction except in the body of a method whose signature indicates it accepts a variable number of arguments. Within such a method its use is verifiable, but the verifer requires that the result be treated as a specific value type which is private to the **System.ArgIterator** class.

beq.<length> – branch on equal

Format	**Assembly Format**	**Description**
3B <**I4**>	beq *target*	branch to *target* if equal
2E <**I1**>	beq.s *target*	branch to *target* if equal, short form

Stack Transition:

```
…, value1, value2
```

…

Description:

The **beq** instruction transfers control to *target* if *value1* is equal to *value2*. The effect is identical to performing a **ceq** instruction followed by a **brtrue *target***. *Target* is represented as a signed offset (4 bytes for **beq**, 1 byte for **beq.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

bge.<length> – branch on greater than or equal to

Format	Assembly Format	Description
3C <**I4**>	bge *target*	branch to *target* if greater than or equal to
2F <**I1**>	bge.s *target*	branch to *target* if greater than or equal to, short form

Stack Transition:

```
…, value1, value2



…
```

Description:

The **bge** instruction transfers control to *target* if *value1* is greater than or equal to *value2*. The effect is identical to performing a **clt** instruction followed by a **brfalse** ***target***. *Target* is represented as a signed offset (4 bytes for **bge**, 1 byte for **bge.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

bge.un.<length> – branch on greater than or equal to, unsigned or unordered

Format	Assembly Format	Description
41 <**I4**>	bge.un *target*	branch to *target* if greater than or equal to (unsigned or unordered)
34 <**I1**>	bge.un.s *target*	branch to *target* if greater than or equal to (unsigned or unordered), short form

Stack Transition:

```
…, value1, value2
```

…

Description:

The **bge.un** instruction transfers control to *target* if *value1* is greater than or equal to *value2,* when compared unsigned (for integer values) or unordered (for float point values). The effect is identical to performing a **clt.un** instruction followed by a **brfalse** ***target***. *Target* is represented as a signed offset (4 bytes for **bge.un**, 1 byte for **bge.un.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

bgt.<length> – branch on greater than

Format	Assembly Format	Description
3D <**I4**>	bgt *target*	branch to *target* if greater than
30 <**I1**>	bgt.s *target*	branch to *target* if greater than, short form

Stack Transition:

```
…, value1, value2
```

…

Description:

The **bgt** instruction transfers control to *target* if *value1* is greater than *value2*. The effect is identical to performing a **cgt** instruction followed by a **brtrue *target***. *Target* is represented as a signed offset (4 bytes for **bgt**, 1 byte for **bgt.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

bgt.un.<length> – branch on greater than, unsigned or unordered

Format	**Assembly Format**	**Description**
42 <**I4**>	bgt.un *target*	branch to *target* if greater than (unsigned or unordered)
35 <**I1**>	bgt.un.s *target*	branch to *target* if greater than (unsigned or unordered), short form

Stack Transition:

```
…, value1, value2
```

…

Description:

The **bgt.un** instruction transfers control to *target* if *value1* is greater than *value2,* when compared unsigned (for integer values) or unordered (for float point values). The effect is identical to performing a **cgt.un** instruction followed by a **brtrue** ***target***. *Target* is represented as a signed offset (4 bytes for **bgt.un**, 1 byte for **bgt.un.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

ble.<length> – branch on less than or equal to

Format	Assembly Format	Description
3E <**I4**>	ble *target*	branch to *target* if less than or equal to
31 <**I1**>	ble.s *target*	branch to *target* if less than or equal to, short form

Stack Transition:

```
…, value1, value2



…
```

Description:

The **ble** instruction transfers control to *target* if *value1* is less than or equal to *value2*. The effect is identical to performing a **cgt** instruction followed by a **brfalse** ***target***. *Target* is represented as a signed offset (4 bytes for **ble**, 1 byte for **ble.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

ble.un.<length> – branch on less than or equal to, unsigned or unordered

Format	Assembly Format	Description
43 <**I4**>	ble.un *target*	branch to *target* if less than or equal to (unsigned or unordered)
36 <**I1**>	ble.un.s *target*	branch to *target* if less than or equal to (unsigned or unordered), short form

Stack Transition:

```
…, value1, value2
```

…

Description:

The **ble.un** instruction transfers control to *target* if *value1* is less than or equal to *value2,* when compared unsigned (for integer values) or unordered (for float point values). The effect is identical to performing a **cgt.un** instruction followed by a **brfalse *target***. *Target* is represented as a signed offset (4 bytes for **ble.un**, 1 byte for **ble.un.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

blt.<length> – branch on less than

Format	Assembly Format	Description
3F <**I4**>	blt *target*	branch to *target* if less than
32 <**I1**>	blt.s *target*	branch to *target* if less than, short form

Stack Transition:

```
…, value1, value2
```

…

Description:

The **blt** instruction transfers control to *target* if *value1* is less than *value2*. The effect is identical to performing a **clt** instruction followed by a **brtrue** ***target***. *Target* is represented as a signed offset (4 bytes for **blt**, 1 byte for **blt.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

blt.un.<length> – branch on less than, unsigned or unordered

Format	Assembly Format	Description
44 <**I4**>	blt.un *target*	branch to *target* if less than (unsigned or unordered)
37 <**I1**>	blt.un.s *target*	branch to *target* if less than (unsigned or unordered), short form

Stack Transition:

```
…, value1, value2
```

…

Description:

The **blt.un** instruction transfers control to *target* if *value1* is less than *value2*. The effect is identical to performing a **clt.un** instruction followed by a **brtrue** ***target***. *Target* is represented as a signed offset (4 bytes for **blt.un**, 1 byte for **blt.un.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

bne.un<length> – branch on not equal or unordered

Format	Assembly Format	Description
40 <**I4**>	bne.un *target*	branch to *target* if unequal or unordered
33 <**I1**>	bne.un.s *target*	branch to *target* if unequal or unordered, short form

Stack Transition:

```
…, value1, value2
```

…

Description:

The **bne.un** instruction transfers control to *target* if *value1* is not equal to *value2,* when compared unsigned (for integer values) or unordered (for float point values). The effect is identical to performing a **ceq** instruction followed by a **brfalse** ***target***. *Target* is represented as a signed offset (4 bytes for **bne.un**, 1 byte for **bne.un.s**) from the beginning of the instruction following the current instruction.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee that the top two items on the stack correspond to the types shown in Table 4: Binary Comparison or Branch Operations.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

br.<length> – unconditional branch

Format	**Assembly Format**	**Description**
38 <**I4**>	br *target*	branch to *target*
2B <**I1**>	br.s *target*	branch to *target*, short form

Stack Transition:

…,

…

Description:

The **br** instruction unconditionally transfers control to *target*. *Target* is represented as a signed offset (4 bytes for **br**, 1 byte for **br.s**) from the beginning of the instruction following the current instruction.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Rationale: *While a **leave** instruction can always be used instead of a **br** instruction, doing so may increase the resources required to compile from IL to native code and/or lead to inferior native code. Therefore IL generators should use a **br** instruction in preference to a **leave** instruction when both are legal.*

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

break – breakpoint instruction

Format	Assembly Format	Description
01	break	inform a debugger that a breakpoint has been reached.

Stack Transition:

…,

…

Description:

The **break** instruction is for debugging support. It signals the EE to inform the debugger that a break point has been tripped. It has no other effect on the interpreter state.

The **break** instruction has the smallest possible instruction size so that code can be patched with a breakpoint with minimal disturbance to the surrounding code.

Exceptions:

None.

Verifiability:

The **break** instruction is always verifiable.

brfalse.<length> - branch on false, null, or zero

Format	Assembly Format	Description
39 <**I4**>	brfalse *target*	branch to *target* if *value* is zero (false)
2C <**I1**>	brfalse.s *target*	branch to *target* if *value* is zero (false), short form
39 <**I4**>	brnull *target*	branch to *target* if *value* is null (*alias for* **brfalse**)
2C <**I1**>	brnull.s *target*	branch to *target* if *value* is null (*alias for* **brfalse.s**), short form
39 <**I4**>	brzero *target*	branch to *target* if *value* is zero (*alias for* **brfalse**)
2C <**I1**>	brzero.s *target*	branch to *target* if *value* is zero (*alias for* **brfalse.s**), short form

Stack Transition:

```
…, value
```

…

Description:

The **brfalse** instruction transfers control to *target* if *value* (of any integral type) is zero (false). If *value* is non-zero (true) execution continues at the next instruction.

If the *value* is an object reference (type **O**), a managed pointer (type **&**) or transient pointer (type *****), then **brnull** (an alias for **brfalse**) transfers control if it represents the null object (see **ldnull**).

Target is represented as a signed offset (4 bytes for **brfalse**, 1 byte for **brfalse.s**) from the beginning of the instruction following the current instruction.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee there is a minimum of one item on the stack.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

brtrue.<length> - branch on non-false or non-null

Format	Assembly Format	Description
3A <**I4**>	brtrue *target*	branch to *target* if *value* is non-zero (true)
2D <**I1**>	brtrue.s *target*	branch to *target* if *value* is non-zero (true), short form
3A <**I4**>	brinst *target*	branch to *target* if *value* is a non-null object reference (alias for **brtrue**)
2D <**I1**>	brinst.s *target*	branch to *target* if *value* is a non-null object reference, short form (*alias for* **brtrue.s**)

Stack Transition:

```
…, value
```

…

Description:

The **brtrue** instruction transfers control to *targe*t if *value* (of any integral type) is nonzero (true). If *value* is zero (false) execution continues at the next instruction.

If the *value* is an object reference (type **O**) then **brinst** (an alias for **brtrue**) transfers control if it represents an instance of an object (i.e. isn't the null object reference, see **ldnull**).

Target is represented as a signed offset (4 bytes for **brtrue**, 1 byte for **brtrue.s**) from the beginning of the instruction following the current instruction.

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL must observe all of the control transfer rules specified above and must guarantee there is a minimum of one item on the stack.

In addition, verifiable code requires the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the verifier specification for more details.

call – call a method

Format	Assembly Format	Description
28 <T>	call *method*	Call method described by *method*

Stack Transition:

```
…, arg1, arg2 … argn



…, retVal (not always returned)
```

Description:

The **call** instruction calls the method indicated by the descriptor *method*. *Method* is a metadata token (either a **methodref** or **methoddef,** see the Metadata Specification) that indicates the method to call and the number, type, and order of the arguments that have been placed on the stack to be passed to that method as well as the calling convention to be used. See the EE Architecture Specification for a detailed description of the IL calling sequence. The **call** instruction may be immediately preceded by a **tail.** prefix to specify that the current method state should be released before transferring control. If the call would transfer control to a method of higher trust than the origin method the stack frame will not be released; instead, the execution will continue silently as if the **tail.** prefix had not been supplied.

The metadata token carries sufficient information to know whether the call is to a static method, an instance method, or a global function. In all of these cases the destination address is determined from the metadata token (See the **callvirt** instruction for calling virtual methods or methods on interfaces).

The arguments are placed on the stack in left-to-right order. That is, the first argument is computed and placed on the stack, then the second argument, etc. There are three important special cases:

1. Calls to an instance (or virtual, see below) method must push that instance reference (the **this** pointer) before any of the user-visible arguments. The **this** pointer must not be null. The signature carried in the metadata does not contain an entry in the parameter list for the **this** pointer but uses a bit to indicate whether the method requires passing the **this** pointer (see the Metadata Specification).
2. It is legal to call a virtual method using **call** (rather than **callvirt**); this indicates that the method is to be resolved using the class specified by *method* rather than as specified dynamically from the object being invoked. This is used, for example, to compile calls to “methods on **super”** (i.e. the statically known parent class).
3. The **call** instruction may not be used to call a delegate’s **Invoke** method: **callvirt** must be used instead.

Exceptions:

SecurityException may be thrown if the runtime system security does not grant the caller access to the called method. The security check may occur when the IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that the stack contains the correct number and type of arguments for the method being called.

For a typical use of the **call** instruction, the verifier checks that (a) *method* refers to a valid **methodref** or **methoddef** token; (b) the types of the objects on the stack are consistent with the types expected by the method call, and (c) the method is accessible from the callsite.

The **call** instruction may also be used to call an object's superclass constructor, or to initialize a value type location by calling an appropriate constructor, both of which are treated as special cases by the verifier. A **call** annotated by **tail.** is also a special case.

calli– indirect method call

Format	Assembly Format	Description
29 <T>	calli *callsitedescr*	Call method indicated on the stack with arguments described by *callsitedescr*.

Stack Transition:

```
…, arg1, arg2 … argn, ftn


… retVal (not always returned)
```

Description:

The **calli** instruction calls *ftn* (a pointer to a method entry point) with the arguments **arg1 … argn**. The types of these arguments are described by the signature **callsitedescr**. See the EE Architecture Specification for a description of the IL calling sequence. The **calli** instruction may be immediately preceded by a **tail.** prefix to specify that the current method state should be released before transferring control. If the call would transfer control to a method of higher trust than the origin method the stack frame will not be released; instead, the execution will continue silently as if the **tail.** prefix had not been supplied.

The *ftn* argument is assumed to be a pointer to native code (of the target machine) that can be legitimately called with the arguments described by *callsitedescr* (a metadata token for a stand-alone signature, see the Metadata Specification). Such a pointer can be created using the **ldftn** or **ldvirtftn** instructions, or have been passed in from native code.

The standalone signature specifies the number and type of parameters being passed, as well as the calling convention (see the Metadata Specification). The calling convention is not checked dynamically, so code that uses a **calli** instruction will not work correctly if the destination does not actually use the specified calling convention.

The arguments are placed on the stack in left-to-right order. That is, the first argument is computed and placed on the stack, then the second argument, etc. The argument-building code sequence for an instance or virtual method must push that instance reference (the **this** pointer, which must not be null) before any of the user-visible arguments.

Exceptions:

SecurityException may be thrown if the runtime system security does not grant the caller access to the called method. The security check may occur when the IL is converted to native code rather than at runtime.

Verifiability:

Correct IL requires that the function pointer contains the address of a method whose signature matches that specified by *callsitedescr* and that the arguments correctly correspond to the types of the destination function's parameters.

The verifier checks that *ftn* is a pointer to a function generated by **ldftn** or **ldvirtfn**. Since these pointers may not be passed as values in verifiable code, this means that the pointer must have been generated somewhere in the current method body.

ceq - compare equal

Format	Assembly Format	Description
FE 01	ceq	push 1 (of type **I4**) if *value*1 equals *value*2, else 0

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The **ceq** instruction compares *value*1 and *value*2. If *value*1 is equal to *value*2, then 1 (of type **I4**) is pushed on the stack. Otherwise 0 (of type **I4**) is pushed on the stack.

For floating point number, **ceq** will return 0 if the numbers are unordered (either or both are NaN). The infinite values are equal to themselves.

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct IL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

cgt - compare greater than

Format	Assembly Format	Description
FE 02	cgt	push 1 (of type **I4**) if *value*1 > *value*2, else 0

Stack Transition:

```
…, value1, value2


…, result
```

Description:

The **cgt** instruction compares *value1* and *value2*. If *value1* is strictly greater than *value2*, then 1 (of type **I4**) is pushed on the stack. Otherwise 0 (of type **I4**) is pushed on the stack

For floating point numbers, **cgt** returns 0 if the numbers are unordered (that is, if one or both of the arguments are NaN).

As per IEEE 754 spec, infinite values are ordered with respect to normal numbers (e.g +infinity > 5.0 > -infinity).

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct IL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

cgt.un - compare greater than, unsigned or unordered

Format	Assembly Format	Description
FE 03	cgt.un	push 1 (of type **I4**) if *value*1 > *value*2, unsigned or unordered, else 0

Stack Transition:

```
…, value1, value2


…, result
```

Description:

The **cgt.un** instruction compares *value1* and *value2*. A value of 1 (of type **I4**) is pushed on the stack if any of the following is true:

- for floating point numbers, *value1* is not ordered with respect to *value2*
- for integer values, *value1* is strictly greater than *value2* when considered as unsigned numbers

Otherwise 0 (of type **I4**) is pushed on the stack.

As per IEEE 754 spec, infinite values are ordered with respect to normal numbers (e.g +infinity > 5.0 > -infinity).

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct IL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

ckfinite – check for a finite real number

Format	Assembly Format	Description
C3	ckfinite	throw **ArithmeticException** if value is not a finite number

Stack Transition:

```
…, value

…, value
```

Description:

The **ckfnite** instruction throws **ArithmeticException** if *value* (a floating point number) is either a "not a number" value (NaN) or +- infinity value. **Ckfinite** leaves the value on the stack if no exception is thrown. Execution is unspecified if *value* is not a floating point number.

Exceptions:

ArithmeticException is thrown if *value* is not a 'normal' number.

> **Note:** A special exception or a subclass of **ArithmeticException** may be more appropriate so that the offending value can be passed to the exception handler.

Verifiability:

Correct IL guarantees that *value* is a floating-point number. There are no additional verification requirements.

clt - compare less than

Format	Assembly Format	Description
FE 04	clt	push 1 (of type **I4**) if *value*1 < *value*2, else 0

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The **clt** instruction compares *value1* and *value2*. If *value1* is strictly less than *value2*, then 1 (of type **I4**) is pushed on the stack. Otherwise 0 (of type **I4**) is pushed on the stack

For floating point numbers, **clt** will return 0 if the numbers are unordered (that is one or both of the arguments are NaN).

As per IEEE 754 spec, infinite values are ordered with respect to normal numbers (e.g +infinity > 5.0 > -infinity).

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct IL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

clt.un - compare less than, unsigned or unordered

Format	Assembly Format	Description
FE 05	clt.un	push 1 (of type **I4**) if *value*1 < *value2*, unsigned or unordered, else 0

Stack Transition:

```
…, value1, value2


…, result
```

Description:

The **clt.un** instruction compares *value1* and *value2*. A value of 1 (of type **I4**) is pushed on the stack if any of the following is true:

- *value1* is strictly less than *value2* (as for **clt**)
- for floating point numbers, *value1* is not ordered with respect to *value2*
- for integer values, *value1* is strictly less than *value2* when considered as unsigned numbers

Otherwise 0 (of type **I4**) is pushed on the stack.

Unlike **clt, clt.un** returns 1 if the numbers are unordered (that is, if one or both of the arguments are NaN).

As per IEEE 754 spec, infinite values are ordered with respect to normal numbers (e.g +infinity > 5.0 > -infinity).

The acceptable operand types are encapsulated in Table 4: Binary Comparison or Branch Operations.

Exceptions:

None.

Verifiability:

Correct IL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations. There are no additional verification requirements.

conv.<to type> - data conversion

Format	**Assembly Format**	**Description**
67	conv.i1	Convert to **I1**, pushing **I4** on stack
68	conv.i2	Convert to **I2**, pushing **I4** on stack
69	conv.i4	Convert to **I4**, pushing **I4** on stack
6A	conv.i8	Convert to **I8**, pushing **I8** on stack
6B	conv.r4	Convert to **R4**, pushing **F** on stack
6C	conv.r8	Convert to **R8**, pushing **F** on stack
D2	conv.u1	Convert to **U1**, pushing **I4** on stack
D1	conv.u2	Convert to **U2**, pushing **I4** on stack
6D	conv.u4	Convert to **U4**, pushing **I4** on stack
6E	conv.u8	Convert to **U8**, pushing **I8** on stack
D3	conv.i	Convert to **I**, pushing **I** on stack
E0	conv.u	Convert to **U**, pushing **I** on stack
76	conv.r.un	Convert unsigned integer to floating point, pushing **F** on stack

Stack Transition:

```
…, value

…, result
```

Description:

Convert the value on top of the stack to the type specified in the opcode, and leave that converted value on the top of the stack. Note that integer value of 4 bytes or less are extended to **I4** (not **I**) when they are loaded onto the evaluation stack, and floating point values are converted to the **F** type.

Conversion from floating point numbers to integral values truncates the number toward zero. When converting from an R8 to an R4, precision may be lost. If *value* is too large to fit in an R4, the IEEE positive infinity (if *value* is positive) or IEEE negative infinity (if *value* is negative) is returned. If overflow occurs converting one integer type to another the high order bits are silently truncated. If the result is smaller than an I4, then the value is sign-extended to fill the slot.

If overflow occurs converting a floating-point type to an integer the value returned is unspecified. The **conv.r.un** operation takes an unsigned integer off of the stack and replaces it with a floating-point number with precision sufficient to represent the integer exactly where possible, or type RPrecise (see Section 1.1.1).

No exceptions are ever thrown. See **conv.ovf** for instructions that will throw an exception when the result type can not properly represent the result value.

The acceptable operand types and their corresponding result data type is encapsulated in Table 7: Conversion Operations.

Exceptions:

None.

Verifiability:

Correct IL has at least one value, of a type specified in Table 7: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

conv.ovf.<to type> - data conversion with overflow detection

Format	Assembly Format	Description
B3	conv.ovf.i1	Convert to an **I1** (on the stack as **I4**) and throw an exception on overflow
B5	conv.ovf.i2	Convert to an **I2** (on the stack as **I4**) and throw an exception on overflow
B7	conv.ovf.i4	Convert to an **I4** (on the stack as **I4**) and throw an exception on overflow
B9	conv.ovf.i8	Convert to an **I8** (on the stack as **I8**) and throw an exception on overflow
B4	conv.ovf.u1	Convert to a **U1** (on the stack as **I4**) and throw an exception on overflow
B6	conv.ovf.u2	Convert to a **U2** (on the stack as **I4**) and throw an exception on overflow
B8	conv.ovf.u4	Convert to a **U4** (on the stack as **I4**) and throw an exception on overflow
BA	conv.ovf.u8	Convert to a **U8** (on the stack as **I8**) and throw an exception on overflow
D4	conv.ovf.i	Convert to an **I** (on the stack as **I**) and throw an exception on overflow
D5	conv.ovf.u	Convert to a **U** (on the stack as **I**) and throw an exception on overflow

Stack Transition:

```
…, value
```

```
…, result
```

Description:

Convert the value on top of the stack to the type specified in the opcode, and leave that converted value on the top of the stack. If the value cannot be represented, an exception is thrown.

Conversions from floating point numbers to integral values truncate the number toward zero. Note that integer value of 4 bytes or less are extended to **I4** (not **I**) on the evaluation stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 7: Conversion Operations.

Exceptions:

OverflowException is thrown if the result can not be represented in the result type

Verifiability:

Correct IL has at least one value, of a type specified in Table 7: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

conv.ovf.<to type>.un – unsigned data conversion with overflow detection

Format	Assembly Format	Description
82	conv.ovf.i1.un	Convert unsigned to an **I1** (on the stack as **I4**) and throw an exception on overflow
83	conv.ovf.i2.un	Convert to unsigned an **I2** (on the stack as **I4**) and throw an exception on overflow
84	conv.ovf.i4.un	Convert to unsigned an **I4** (on the stack as **I4**) and throw an exception on overflow
85	conv.ovf.i8.un	Convert to unsigned an **I8** (on the stack as **I8**) and throw an exception on overflow
86	conv.ovf.u1.un	Convert to unsigned a **U1** (on the stack as **I4**) and throw an exception on overflow
87	conv.ovf.u2.un	Convert to unsigned a **U2** (on the stack as **I4**) and throw an exception on overflow
88	conv.ovf.u4.un	Convert to unsigned a **U4** (on the stack as **I4**) and throw an exception on overflow
89	conv.ovf.u8.un	Convert to unsigned a **U8** (on the stack as **I8**) and throw an exception on overflow
8A	conv.ovf.i.un	Convert to unsigned an **I** (on the stack as **I**) and throw an exception on overflow
8B	conv.ovf.u.un	Convert to unsigned a **U** (on the stack as **I**) and throw an exception on overflow

Stack Transition:

```
…, value
```

```
…, result
```

Description:

Convert the value on top of the stack to the type specified in the opcode, and leave that converted value on the top of the stack. If the value cannot be represented, an exception is thrown. The item at the top of the stack is treated as an unsigned value.

Conversions from floating point numbers to integral values truncate the number toward zero. Note that integer value of 4 bytes or less are extended to **I4** (not **I**) on the evaluation stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 7: Conversion Operations.

Exceptions:

OverflowException is thrown if the result can not be represented in the result type

Verifiability:

Correct IL has at least one value, of a type specified in Table 7: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

cpblk - copy data from memory to memory

Format	Instruction	Description
FE 17	cpblk	Copy data from memory to memory

Stack Transition:

```
…, destaddr, srcaddr, size
```

…

Description:

The **cpblk** instruction copies *size* (of type U4) bytes from address *srcaddr* (of type *, **I,** or **&**) to address *destaddr* (of type ***, I,** or **&**). The behavior of **cpblk** is unspecified if the source and destination areas overlap.

cpblk assumes that both *destaddr* and *srcaddr* are aligned to the natural size of the machine (but see the **unaligned.** prefix instruction). The **cpblk** instruction may be immediately preceded by the **unaligned.** prefix instruction to indicate that either the source or the destination is unaligned. It is an appropriate expansion of methods like C's **memcpy**.

The operation of the **cpblk** instruction may be altered by an immediately preceding **volatile.** or **unaligned.** prefix instruction.

Exceptions:

InvalidAddressException may be thrown if an invalid address is detected.

Verifiability:

The **cpblk** instruction is never verifiable. Correct IL ensures the conditions specified above.

div - divide values

Format	Assembly Format	Description
5B	div	Divide two values to return a quotient or floating point result

Stack Transition:

```
…, value1, value2


…, result
```

Description:

The div instruction computes *value1* divided by *value2* and pushes the result on the stack. The type of the values and result are the same.

Floating point division is per IEE754 spec. In particular division of a finite number by 0 produces the correctly signed infinite value and

0 / 0 = **NaN**

infinity / **infinity** = **NaN**.

X / **infinity** = 0

The acceptable operand types and their corresponding result data type is encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

Integral operations throw **ArithmeticException** if the result can not be represented in the result type. This can happen if *value1* is the maximum negative value, and *value2* is -1.

Integral operations throw **DivideByZeroException** if *value2* is zero.

> **Note:** On the x86 an **ArithmeticOverflowException** is thrown when computing (minint div –1).

Floating-point operations never throw an exception (they produce NaNs instead, see the EE Architecture Specification).

Verifiability:

See Table 2: Binary Numeric Operations**.**

div.un - divide integer values, unsigned

Format	Assembly Format	Description
5C	div.un	Divide two values, unsigned, returning a quotient

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The div instruction computes *value1* divided by *value2,* both taken as unsigned integers, and pushes the result on the stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5.

Exceptions:

DivideByZeroException is thrown if *value2* is zero.

Verifiability:

See Table 5: Integer Operations.

dup – duplicate the top value of the stack

Format	Assembly Format	Description
25	dup	duplicate value on the top of the stack

Stack Transition:

```
…, value


…, value, value
```

Description:

The **dup** instruction duplicates the top element of the stack.

Exceptions:

None.

Verifiability:

No additional requirements.

endfilter – end filter clause of SEH

Format	Assembly Format	Description
FE 11	endfilter	End filter clause of SEH exception handling

Stack Transition:

```
…, value
```

…

Description:

Return from **filter** clause of an exception (see the Exception Handling section of the EE Architecture Specification for an overview or the Exception Specification for details). *Value* (which must be of type I4 and is one of a specific set of values) is returned from the **filter** clause. It should be one of:

- **exception_continue_execution** (-1) to continue execution at the instruction after the one which raised the exception (currently treated as 0)
- **exception_continue_search** (0) to continue searching for an exception handler
- **exception_execute_handler** (1) to start the second phase of exception handling where finally blocks are run until the handler associated with this filter clause is located. Then the handler is executed.

Other integer values will produce unspecified results.

The entry point of a filter, as shown in the method's exception table, must be the (lexically) first instruction in the filter's code block. The **endfilter** must be the (lexically) last instruction in the filter's code block (hence there can only be one **endfilter** for any single filter block). After executing the **endfilter** instruction, control logically flows back to the runtime exception handling mechanism.

Control cannot be transferred into a **filter** block except through the exception mechanism. Control cannot be transferred out of a **filter** block except through the use of a **throw** instruction or executing the final **endfilter** instruction. In particular, it is not legal to execute a **ret** or **leave** instruction within a **finally** block. It is not legal to embed a **try** block within a **filter** block.

Exceptions:

None.

Verifiability:

Correct IL guarantees the control transfer restrictions specified above. There are no additional verification requirements.

endfinally – end finally clause of an exception block

Format	Assembly Format	Description
DC	endfinally	End finally clause of an exception block

Stack Transition:

…

…

Description:

Return from **finally** clause of an exception block, see the Exception Handling section of the EE Architecture Specification or the Exception Specification for details. Signals the end of the **finally** clause so that stack unwinding can continue until the exception handler is invoked. The **endfinally** instruction transfers control back to the Runtime exception mechanism. This then searches for the next **finally** clause in the exception chain or enters the exception handler chosen during the first pass of exception handling.

An **endfinally** instruction may only appear lexically within a **finally** block. Unlike the **endfilter** instruction, there is no requirement that the block end with an **endfinally** instruction, and there can be as many **endfinally** instructions within the block as required.

Control cannot be transferred into a **finally** block except through the exception mechanism. Control cannot be transferred out of a **finally** block except through the use of a **throw** instruction or executing the **endfinally** instruction. In particular, it is not legal to “fall out” of a **finally** block or to execute a **ret** or **leave** instruction within a **finally** block.

Exceptions:

None.

Verifiability:

Correct IL guarantees the control transfer restrictions specified above. There are no additional verification requirements.

initblk - initialize a block of memory to a value

Format	Assembly Format	Description
FE 18	initblk	Set a block of memory to a given byte

Stack Transition:

```
…, addr, value, size
```

…

Description:

The **initblk** instruction sets *size* (of type **U4**) bytes starting at *addr* (of type **I, &**, or *****) to *value* (of type U1). **initblk** assumes that *addr* is aligned to the natural size of the machine (but see the **unaligned.** prefix instruction). It is an appropriate expansion of methods like C's **memset**.

The operation of the **initblk** instructions may be altered by an immediately preceding **volatile.** or **unaligned.** prefix instruction.

Exceptions:

InvalidAddressException may be thrown if an invalid address is detected.

Verifiability:

The **initblk** instruction is never verifiable. Correct IL code ensures the restrictions specified above.

jmp – jump to method

Format	**Assembly Format**	**Description**
27 <**T**>	jmp *method*	Exit current method and jump to specified method

Stack Transition:

…

…

Description:

Transfer control to the method specified by *method*, which is a metadata token (either a methodref or methoddef, see the Metadata Specification). The current arguments are transferred to the destination method.

The evaluation stack must be empty when this instruction is executed. The calling convention, number and type of arguments at the destination address must match that of the current method.

The jmp instruction cannot be used to transferred control out of a try, filter, catch, or finally block. See the Exception Specification .

Exceptions:

None.

Verifiability:

The **jmp** instruction is never verifiable. Correct IL code obeys the control flow restrictions specified above.

jmpi – jump via method pointer

Format	Assembly Format	Description
FE 08	jmpi	Exit current method and jump to specified method

Stack Transition:

```
… ftn
```

…

Description:

Transfer control to the method specified by *ftn*, which must be a pointer to native code (of the target machine) that can be legitimately called with the arguments to the current method. Such a pointer can be created using the ldftn or ldvirtftn instructions, or have been passed in from native code. The stack must be empty except for the *ftn* when this instruction is executed; the verifier checks this.

The current method signature specifies the number and type of parameters being passed, as well as the calling convention (see the Metadata Specification). The calling convention is not checked dynamically, so code that uses a jmpi instruction will not work correctly if the destination does not actually use the specified calling convention.

The jmpi instruction cannot be used to transferred control out of a try, filter, catch, or finally block. See the Exception Specification.

***Rationale:** The **jmpi** instruction is used in a stylized manner for C++ adjustor thunks. It is also useful as an optimization of the **tail. calli** sequence for certain operations in functional programming languages.*

Exceptions:

None.

Verifiability:

The **jmpi** instruction is never verifiable. Correct IL code obeys the control flow restrictions specified above.

ldarg.<length> - load argument onto the stack

Format	Assembly Format	Description
FE 09 <**U2**>	ldarg *num*	Load argument numbered *num* onto stack.
0E <**U1**>	ldarg.s *num*	Load argument numbered *num* onto stack, short form.
02	ldarg.0	Load argument 0 onto stack
03	ldarg.1	Load argument 1 onto stack
04	ldarg.2	Load argument 2 onto stack
05	ldarg.3	Load argument 3 onto stack

Stack Transition:

…

```
…, value
```

Description:

The **ldarg** *num* instruction pushes the incoming argument numbered *num* (see the EE Architecture Specification) onto the evaluation stack. The **ldarg** instruction can be used to load a value type or a primitive value onto the stack by copying it from an incoming argument. The type of the value is the same as the type of the argument, as specified by the current method's signature.

The **ldarg.0**, **ldarg.1**, **ldarg.2**, and **ldarg.3** instructions are efficient encodings for loading any one of the first 4 arguments. The **ldarg.s** instruction is an efficient encoding for loading the 5th through 256th argument.

For procedures that take a variable-length argument list, the **ldarg** instructions can be used only for the initial fixed arguments, not those in the variable part of the signature.

Arguments that hold an integer value smaller than 4 bytes long are expanded to type I4 when they are loaded onto the stack. Floating-point values are expanded to their native size (type **F**).

Exceptions:

None.

Verifiability:

Correct IL guarantees that *num* is a valid argument index. See the Verifier Specification for more details on how the verifier determines the type of the value loaded onto the stack.

ldarga.<length> - load an argument address

Format	**Assembly Format**	**Description**
FE 0A <**U2**>	ldarga *argNum*	fetch the address of argument *argNum*.
0F <**U1**>	ldarga.s *argNum*	fetch the address of argument *argNum*, short form

Stack Transition:

```
…,
```

```
…, address of argument number argNum
```

Description:

The ldarga instruction fetches the address (of type *, i.e. transient pointer) of argument *argNum*. The address will always be aligned to a natural boundary on the target machine (cf. cpblk and initblk). The short form (ldarga.s) should be used for arguments 0 through 255.

For procedures that take a variable-length argument list, the ldarga instructions can be used only for the initial fixed arguments, not those in the variable part of the signature.

***Rationale: ldarga** is used for by-ref parameter passing (see the EE Architecture Specification). In other cases, **ldarg** and **starg** should be used.*

Exceptions:

None.

Verifiability:

Correct IL ensures that *argNum* is a valid argument index. See the Verifier Specification for more details.

ldc.<type> - load numeric constant

Format	Assembly Format	Description
20 <**I4**>	ldc.i4 *num*	Push *num* of type **I4** onto the stack as **I4**.
21 <**I8**>	ldc.i8 *num*	Push *num* of type **I8** onto the stack as **I8**.
22 <**R4**>	ldc.r4 *num*	Push *num* of type **R4** onto the stack as **F**.
23 <**R8**>	ldc.r8 *num*	Push *num* of type **R8** onto the stack as **F**.
16	ldc.i4.0	Push 0 onto the stack as **I4**.
17	ldc.i4.1	Push 1 onto the stack as **I4**.
18	ldc.i4.2	Push 2 onto the stack as **I4**.
19	ldc.i4.3	Push 3 onto the stack as **I4**.
1A	ldc.i4.4	Push 4 onto the stack as **I4**.
1B	ldc.i4.5	Push 5 onto the stack as **I4**.
1C	ldc.i4.6	Push 6 onto the stack as **I4**.
1D	ldc.i4.7	Push 7 onto the stack as **I4**.
1E	ldc.i4.8	Push 8 onto the stack as **I4**.
15	ldc.i4.m1	Push -1 onto the stack as **I4**.
15	ldc.i4.M1	Push -1 of type **I4** onto the stack as **I4** (alias for **ldc.i4.m1**).
1F <**I1**>	ldc.i4.s *num*	Push *num* onto the stack as **I4**, short form.

Stack Transition:

…

…, num

Description:

The **ldc** *num* instruction pushes number *num* onto the stack. There are special short encodings for the integers –128 through 127 (with especially short encodings for –1 through 8). All short encodings push 4 byte integers on the stack. Longer encodings are used for 8 byte integers and 4 and 8 byte floating point numbers.

There are three ways to push an 8 byte integer constant onto the stack

1. use the **ldc.i8** instruction for constants that must be expressed in more than 32 bits
2. use the **ldc.i4** instruction followed by a **conv.i8** for constants that require 9 to 32 bits
3. use a short form instruction followed by a **conv.i8** for constants that can be expressed in 8 or fewer bits

There is no way to express a floating point constant that has a larger range or greater precision than a 64 bit IEEE 754 number, since these representations are not portable across architectures.

Exceptions:

None.

Verifiability:

The **ldc** instruction is always verifiable.

ldftn - load method pointer

Format	Assembly Format	Description
FE 06 <**T**>	ldftn *method*	Push a pointer to a method referenced by *method* on the stack

Stack Transition:

…

…, ftn

Description:

The **ldftn** instruction pushes an unmanaged pointer (type **I**) to the native code implementing the method described by *method* (a metadata token, either a **methoddef** or **methodref**, see the Metadata Specification) onto the stack. The value pushed can be called using the **calli** instruction if it references a managed method (or a stub that transitions from managed to unmanaged code).

The value returned points to native code using the calling convention specified by *method*. Thus a method pointer can be passed to unmanaged native code (e.g. as a callback routine). Note that the address computed by this instruction may be to a thunk produced specially for this purpose (for example, to re-enter the IL interpreter when a native version of the method isn't available).

Exceptions:

None.

Verifiability:

Correct IL requires that *method* is a valid **methoddef** or **methodref** token. The verifier tracks the type of the value pushed in more detail than the "**I**" type, remembering that it is a method pointer. Such a method pointer can then be used with **calli** or to construct a delegate.

ldind.<type> - load value indirect onto the stack

Format	Assembly Format	Description
46	ldind.i1	Indirect load value of type **I1** as **I4** on stack.
48	ldind.i2	Indirect load value of type **I2** as **I4** stack.
4A	ldind.i4	Indirect load value of type **I4** as **I4** stack.
4C	ldind.i8	Indirect load value of type **I8** as **I8** stack.
47	ldind.u1	Indirect load value of type **U1** as **I4** stack.
49	ldind.u2	Indirect load value of type **U2** as **I4** stack.
4A	ldind.u4	Indirect load value of type **U4** as **I4** stack. (alias for **ldind.i4**).
4E	ldind.r4	Indirect load value of type **R4** as **F** stack.
4C	ldind.u8	Indirect load value of type **U8** as **I8** stack (alias for **ldind.i8**).
4F	ldind.r8	Indirect load value of type **R8** as **F** stack.
4D	ldind.i	Indirect load value of type **I** as **I** stack
50	ldind.ref	Indirect load value of type **object ref** as **O** on stack.

Stack Transition:

```
…, addr


…, value
```

Description:

The **ldind** instruction indirectly loads a value from address *addr* (an integer, **I**, managed pointer, **&**, or transient pointer, *****) onto the stack. The source value is indicated by the instruction suffix. All of the **ldind** instructions are shortcuts for a **ldobj** instruction that specifies the corresponding built-in value class.

Note that integer value of 4 bytes or less are extended to **I4** (not **I**) when they are loaded onto the evaluation stack. Floating point values are converted to **F** type when loaded onto the evaluation stack.

Correct IL ensures that the **ldind** instructions is used in a manner consistent with the type of the pointer.

The address specified by *addr* must be aligned to the natural size of objects on the machine or an **InvalidAddressException** may occur (but see the **unaligned.** prefix instruction). The results of all IL instructions that return addresses (e.g. **ldloca** and **ldarga)** are safely aligned. For datatypes larger than 1 byte, the byte ordering is dependent on the target CPU. Code that is written that depends on byte ordering may not run on all platforms.

The operation of the **ldind** instructions may be altered by an immediately preceding **volatile.** or **unaligned.** prefix instruction.

Rationale: *Signed and unsigned forms for the small integer types are needed so that the EE can know whether to sign extend or zero extend. The* ***ldind.u8*** *and* ***ldind.u4*** *variants are provided for convenience; they are simply aliases for* ***ldind.i8*** *and* ***ldind.i4*** *respectively.*

Exceptions:

InvalidAddressException may be thrown if an invalid address is detected.

Verifiability:

Correct IL only uses an **ldind** instruction in a manner consistent with the type of the pointer. These instructions cannot be used in verified code.

ldloc - load local variable onto the stack

Format	**Assembly Format**	**Description**
FE 0C<**U2**>	ldloc *indx*	Load local variable of index *indx* onto stack.
11 <**U1**>	ldloc.s *indx*	Load local variable of index *indx* onto stack, short form.
06	ldloc.0	Load local variable 0 onto stack.
07	ldloc.1	Load local variable 1 onto stack.
08	ldloc.2	Load local variable 2 onto stack.
09	ldloc.3	Load local variable 3 onto stack.

Stack Transition:

…

`…, value`

Description:

The **ldloc** *indx* instruction pushes the contents of the local variable with index *indx* onto the evaluation stack. Local variables are initialized to 0 before entering the method only if the initialize flag on the method is true (see the EE Architecture Specification). The **ldloc.0**, **ldloc.1**, **ldloc.2**, and **ldloc.3** instructions provide an efficient encoding for accessing the first four local variables. The **ldloc.s** instruction provides an efficient encoding for accessing the 5th through 256th local variable.

The type of the value is the same as the type of the local variable, which is specified in the method header. See the EE Architecture Specification.

Local variables that are smaller than 4 bytes long are expanded to type I4 when they are loaded onto the stack. Floating-point values are expanded to their native size (type **F**).

Exceptions:

None.

Verifiability:

Correct IL ensures that *indx* is a valid local index. See the Verifier Specification for more details about how the verifier determines the type of local variables.

ldloca.<length> - load local variable address

Format	Assembly Format	Description
FE 0D <**U2**>	ldloca *index*	Load address of local variable with index *indx*
12 <**U1**>	ldloca.s *index*	Load address of local variable with index *indx, short form*

Stack Transition:

…

…, address

Description:

The **ldloca** instruction pushes the address of the local variable with *index* onto the stack. The value pushed on the stack is already aligned correctly for use with instructions like **ldind** and **stind**. The result is a transient pointer (type *). The **ldloca.s** instruction provides an efficient encoding for use with the first 256 local variables.

Exceptions:

None.

Verifiability:

Correct IL ensures that *indx* is a valid local index. See the Verifier Specification for more details on how the verifier determines the type of a local variable.

ldnull – load a null pointer

Format	Assembly Format	Description
14	ldnull	Push null GC reference on the stack

Stack Transition:

…

`…, null value`

Description:

The **ldnull** pushes a null reference (type **O**) on the stack. It is not legal to dereference this null value. They are used to initialize locations before they become live or when they become dead.

Exceptions:

None.

Verifiability:

The **ldnull** instruction is always verifiable, and produces a value that the verifier considers compatible with any other reference type.

leave.<length> – exit a protected region of code

Format	**Assembly Format**	**Description**
DD <**I4**>	leave *target*	Exit a protected region of code.
DE <**I1**>	leave.s *target*	Exit a protected region of code, *short form*

Stack Transition:

…,

…,

Description:

The **leave** instruction unconditionally transfers control to *target*. *Target* is represented as a signed offset (4 bytes for **leave**, 1 byte for **leave.s**) from the beginning of the instruction following the current instruction.

The **leave** instruction is similar to the **br** instruction, but it can be used to exit a **try**, **filter**, or **catch** block whereas the ordinary branch instructions can only be used to transfer control within such a block. It ensures that the appropriate surrounding **finally** blocks are executed.

It is not legal to use a **leave** instruction to exit a **finally** block. To ease code generation for exception handlers it is legal from within a **catch** block to use a **leave** instruction to transfer control to any instruction within the associated **try** block.

If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL requires the destination be within the current method. The verifier will check the type-consistency of the stack, locals and arguments for every possible way of reaching the destination instruction. See the Verifier Specification for more details.

localloc – allocate space in the local dynamic memory pool

Format	Assembly Format	Description
FE 0F	localloc	Allocate space from the local memory pool.

Stack Transition:

```
…, size
```

```
…, address
```

Description:

The **localloc** instruction allocates *size* (type U) bytes from the local dynamic memory pool and returns the address (a transient pointer, type *) of the first allocated byte. The block of memory returned is initialized to 0 only if the initialize flag on the method is true (see the EE Architecture Specification). The area of memory is newly allocated. When the current method returns the local memory pool is available for reuse.

Address is aligned so that any primitive data type can be stored there using the **stind** instructions and loaded using the **ldind** instructions.

The evaluation stack must be empty when this instruction is executed.

***Rationale: Localloc** is used to create local aggregates whose size must be computed at runtime. It can be used for C's intrinsic **alloca** method.*

Exceptions:

ExecutionEngineException is thrown if the stack is not empty at the time this instruction is executed.

StackOverflowException is thrown if there is insufficient memory to service the request.

Verifiability:

Correct IL only uses this instruction within a method that has a non-zero local allocation area size. This instruction is never verifiable.

mul - multiply values

Format	Assembly Format	Description
5A	mul	Multiply values

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The **mul** instruction multiplies *value1* by *value2* and pushes the result on the stack. Integral operations silently truncate the upper bits on overflow (see **mul.ovf**).

For floating point types, 0 * **infinity** = **NaN**.

The acceptable operand types and their corresponding result data type is encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 2: Binary Numeric Operations**.**

mul.ovf.<type> - multiply integer values with overflow check

Format	Assembly Format	Description
D8	mul.ovf	Multiply signed integer values. Signed result must fit in same size
D9	mul.ovf.un	Multiply unsigned integer values. Unsigned result must fit in same size

Stack Transition:

```
…, value1, value2
```

```
…, result
```

Description:

The **mul.ovf** instruction multiplies integers, *value1* and *value2,* and pushes the result on the stack. An exception is thrown if the result will not fit in the result type.

The acceptable operand types and their corresponding result data type is encapsulated in Table 6: Overflow Arithmetic Operations.

Exceptions:

OverflowException is thrown if the result can not be represented in the result type.

Verifiability:

See Table 7: Conversion Operations**.**

neg - negate

Format	**Assembly Format**	**Description**
65	neg	Negate value

Stack Transition:

```
…, value

…, result
```

Description:

The **neg** instruction negates *value* and pushes the result on top of the stack. The return type is the same as the operand type.

Negation of integral values is standard twos complement negation. In particular, negating the most negative number (which does not have a positive counterpart) yields the most negative number. To detect this overflow use the **sub.ovf** instruction instead (i.e. subtract from 0).

Negating a floating point number cannot overflow; negating **NaN** returns **NaN**.

The acceptable operand types and their corresponding result data type is encapsulated in Table 3: Unary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 3: Unary Numeric Operations.

nop – no operation

Format	Assembly Format	Description
00	nop	Do nothing

Stack Transition:

…,

…,

Description:

The **nop** operation does nothing. It is intended to fill in space if bytecodes are patched.

Exceptions:

None.

Verifiability:

The **nop** instruction is always verifiable.

not - bitwise complement

Format	Assembly Format	Description
66	not	Bitwise complement

Stack Transition:

```
…, value



…, result
```

Description:

Compute the bitwise complement of the integer value on top of the stack and leave the result on top of the stack. The return type is the same as the operand type.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

or - bitwise OR

Format	Instruction	Description
60	or	Bitwise OR of two integer values, returns an integer.

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The or instruction computes the bitwise OR of the top two values on the stack and leaves the result on the stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

pop – remove the top element of the stack

Format	Assembly Format	Description
26	pop	pop a value from the stack

Stack Transition:

```
…, value
```

…

Description:

The **pop** instruction removes the top element from the stack.

Exceptions:

None.

Verifiability:

No additional requirements.

rem - compute remainder

Format	**Assembly Format**	**Description**
5D	rem	Remainder of dividing value1 by value2

Stack Transition:

```
…, value1, value2



…, result
```

Description:

result = *value1* ***rem*** *value2* satisfies the following conditions:

$(n*value2)+result = value1$,

where *n* is an integer,

$0 \le |result| < |value2|$, and

$sign(result) = sign(value2)$

The **rem** instruction computes *result* and pushes it on the stack.

For floating point types, **rem** is defined by the IEEE 754 spec. In particular, if *value2* is zero or *value1* is infinity the result is NaN. If *value2* is **infinity**, the result is *value1* (negated for –**infinity**).

The acceptable operand types and their corresponding result data type is encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

Integral operations throw **DivideByZeroException** if *value2* is zero.

> **Note:** On the x86 an **ArithmeticOverflowException** is thrown when computing (minint rem –1).

Example:

+14	rem	+3	is 2	(n = 4)
+14	rem	-3	is -1	(n = 5)
-14	rem	+3	is 1	(n = -5)
-14	rem	-3	is -2	(n = 4)

Verifiability:

See Table 2: Binary Numeric Operations**.**

rem.un - compute integer remainder, unsigned

Format	Assembly Format	Description
5E	rem.un	Remainder of unsigned dividing value1 by value2

Stack Transition:

```
…, value1, value2


…, result
```

Description:

result = *value1* ***rem.un*** *value2* satisfies the following conditions:

$(n*value2)+result = value1$,

where n is an integer,

$0 \leq result < value2$

The **rem.un** instruction computes *result* and pushes it on the stack. **Rem.un** treats its arguments as unsigned integers, while **rem** treats them as signed integers. **Rem.un** is unspecified for floating point numbers.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

The instruction throws **DivideByZeroException** if *value2* is zero.

Note: There is a known bug (5414) that causes JITted code on the x86 to throw a divide overflow when computing (minint rem –1).

Verifiability:

See Table 5: Integer Operations.

ret – return from method

Format	Assembly Format	Description
2A	Ret	Return from method, possibly returning a value

Stack Transition:

```
 retVal     on callee evaluation stack (not always present)



…, retVal   on caller evaluation stack (not always present)
```

Description:

Return from the current method. The return type of the current method determines the type of value to be fetched from the top of the stack and copied onto the stack of the method that called the current method. The evaluation stack for the current method must be empty except for the value to be returned.

The **ret** instruction cannot be used to transfer control out of a **try**, **filter**, **catch**, or **finally** block. From within a **try** or **catch**, use the **leave** instruction with a destination of a **ret** instruction that is outside all enclosing exception blocks. Because the **filter** and **finally** blocks are logically part of exception handling, not the method in which their code is embedded, correctly generate IL does not perform a method return from within a **filter** or **finally**. See the Exception Specification.

Exceptions:

None.

Verifiability:

Correct IL obeys the control constraints describe above. Verification requires that the type of *retVal* is compatible with the declared return type of the current method.

shl - shift integer left

Format	Assembly Format	Description
62	shl	Shift an integer to the left (shifting in zeros)

Stack Transition:

```
…, value, shiftAmount



…, result
```

Description:

The **shl** instruction shifts *value* (an integer) left by the number of bits specified by *shiftAmount*. *shiftAmount* is of type U. The return type is the same as *value*. The return value is unspecified if *shiftAmount* is greater than or equal to the size of *value*.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

shr - shift integer right

Format	Assembly Format	Description
63	shr	Shift an integer right, (shift in sign), return an integer

Stack Transition:

```
…, value, shiftAmount


…, result
```

Description:

The **shr** instruction shifts *value* (an integer) right by the number of bits specified by *shiftAmount*. *shiftAmount* is of type U. The return type is the same as *value*. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. **shr** replicates the high order bit on each shift, preserving the sign of the original value in the result.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

shr.un - shift integer right, unsigned

Format	Assembly Format	Description
64	shr.un	Shift an integer right, (shift in zero), return an integer

Stack Transition:

```
…, value, shiftAmount



…, result
```

Description:

The **shr.un** instruction shifts *value* (an integer) right by the number of bits specified by *shiftAmount*. *shiftAmount* is of type U4. The return type is the same as *value*. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. **Shr.un** inserts a zero bit on each shift.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

starg.<length> - store a value in an argument slot

Format	**Assembly Format**	**Description**
FE 0B <**U2**>	starg *num*	Store a value to the argument numbered *num*
10 <**U1**>	starg.s *num*	Store a value to the argument numbered *num*, short form

Stack Transition:

```
… value
```

…,

Description:

The **starg** *num* instruction pops a value from the stack and places it in argument slot *num* (see the EE Architecture Specification). The type of the value must match the type of the argument, as specified in the current method's signature. The **starg.s** instruction provides an efficient encoding for use with the first 256 arguments.

For procedures that take a variable argument list, the **starg** instructions can be used only for the initial fixed arguments, not those in the variable part of the signature.

Storing into arguments that hold an integer value smaller than 4 bytes long truncates the value as it moves from the stack to the argument. Floating-point values are rounded from their native size (type **F**) to the size associated with the argument.

Exceptions:

None.

Verifiability:

Correct IL requires that *num* is a valid argument slot.

The verifier also checks that the basic EE type of the value matches the basic EE type of the argument, as specified in the current method's signature. The verifier then considers the new verification type of the argument slot to then be the same as the verification type of *value* (verification types are more detailed than EE types). The rules change slightly if the address of the argument slot has been taken prior to this instruction using **ldarga** - see the Verifier Specification for more details.

stind.<type> - store value indirect from stack

Format	Assembly Format	Description
52	stind.i1	Store value of type **I1** into memory at address
53	stind.i2	Store value of type **I2** into memory at address
54	stind.i4	Store value of type **I4** into memory at address
55	stind.i8	Store value of type **I8** into memory at address
56	stind.r4	Store value of type **R4** into memory at address
57	stind.r8	Store value of type **R8** into memory at address
DF	stind.i	Store value of type **I** into memory at address
51	stind.ref	Store value of type **object ref** (type **O**) into memory at address

Stack Transition:

```
…, addr, val
```

…

Description:

The **stind** instruction stores a value *val* at address *addr* (an unmanaged pointer, type **I**, transient pointer, type *****, or managed pointer, type **&**). The address specified by *addr* must be aligned to the natural size of *val* or an **InvalidAddressException** may occur (but see the **unaligned.** prefix instruction). The results of all IL instructions that return addresses (e.g. **ldloca** and **ldarga)** are safely aligned. For datatypes larger than 1 byte, the byte ordering is dependent on the target CPU. Code that is written that depends on byte ordering may not run on all platforms.

Type safe operation requires that the **stind** instruction be used in a manner consistent with the type of the pointer. Code that uses the **stind** instruction cannot be verified except in certain stylized sequences.

The operation of the **stind** instruction may be altered by an immediately preceding **volatile.** or **unaligned.** prefix instruction.

Exceptions:

InvalidAddressException may be thrown if an invalid address is detected.

Verifiability:

Correct IL ensures that *addr* be a pointer whose type is known and is assignment compatible with that of *val*. These instructions are not verifiable.

stloc - pop value from stack to local variable

Format	Assembly Format	Description
FE 0E <**U2**>	stloc *indx*	Pop value from stack into local variable *indx.*
13 <**U1**>	stloc.s *indx*	Pop value from stack into local variable *indx*, short form.
0A	stloc.0	Pop value from stack into local variable 0.
0B	stloc.1	Pop value from stack into local variable 1.
0C	stloc.2	Pop value from stack into local variable 2.
0D	stloc.3	Pop value from stack into local variable 3.

Stack Transition:

```
…, value
```

…

Description:

The **stloc** *indx* instruction pops the top value off the evalution stack and moves it into local variable *indx* (seethe EE Architecture Specification). The type of *value* must match the type of the local variable as specified in the current method's locals signature. The **stloc.0**, **stloc.1**, **stloc.2**, and **stloc.3** instructions provide an efficient encoding for the first four local variables; the **stloc.s** instruction provides an efficient encoding for the 5th through 256th local variables.

Storing into locals that hold an integer value smaller than 4 bytes long truncates the value as it moves from the stack to the argument. Floating-point values are rounded from their native size (type **F**) to the size associated with the argument.

Exceptions:

None.

Verifiability:

Correct IL requires that *indx* is a valid local index.

The verifier also checks that the basic EE type of the value matches the basic EE type of the local, as specified in the current method's locals signature. The verifier then considers the new verification type of the argument slot to then be the same as the verification type of *value* (verification types are more detailed than EE types). The rules change slightly if the address of the local variable has been taken prior to this instruction using **ldloca** - see the Verifier Specification for more details.

sub - subtract numeric values

Format	Assembly Format	Description
59	sub	Subtract *value2* from *value1*, returning a new value

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The **sub** instruction subtracts *value2* from *value1* and pushes the result on the stack. Overflow is not detected for the integral operations (see **sub.ovf**); for floating point operands, **sub** returns **+inf** on positive overflow, **-inf** on negative overflow, and zero on floating point underflow.

The acceptable operand types and their corresponding result data type is encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 2: Binary Numeric Operations**.**

sub.ovf.<type> - subtract integer values, checking for overflow

Format	Assembly Format	Description
DA	sub.ovf	Subtract I from an I. Signed result must fit in same size
DB	sub.ovf.un	Subtract U from a U. Unsigned result must fit in same size

Stack Transition:

```
…, value1, value2



…, result
```

Description:

The **sub.ovf** instruction subtracts *value2* from *value1* and pushes the result on the stack. The type of the values and the return type is specified by the instruction.   An exception is thrown if the result does not fit in the result type.

The acceptable operand types and their corresponding result data type is encapsulated in Table 6: Overflow Arithmetic Operations.

Exceptions:

OverflowException is thrown if the result can not be represented in the result type.

Verifiability:

See Table 6: Overflow Arithmetic Operations.

switch – table switch on value

Format	**Assembly Format**	**Description**
45 <**U4**> <**I4**>… <**I4**>	switch *t1, t2 … tn*	jump to one of n values

Stack Transition:

```
…, value
```

```
…,
```

Description:

The **switch** instruction implements a table jump. The format of the instruction itself is a U4 representing the number of targets **N**, followed by **N** I4 values representing target locations. The targets are represented as offsets from the beginning of the instruction following the switch instruction.

The switch instruction pops *value* (an I4) off the stack and compares it as an unsigned integer to **N.** If *value* is less than **N,** *value* is used as an index into the target array (starting at 0), and execution continues at the selected target. If *value* is not less than **N**, execution continues at the next instruction (fall through).

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by the ordinary branch instructions (use the **leave** instruction instead). These transfers are severely restricted; see the Exception Specification for details. If an instruction has one or more prefix codes, control can only be transferred to the first of these prefixes.

Exceptions:

None.

Verifiability:

Correct IL obeys the control transfer constraints listed above. In addition, verification requires the type-consistency of the stack, locals and arguments for every possible way of reaching all destination instructions. See the verifier specification for more details.

tail. (prefix code) – subsequent call terminates current method

Format	Assembly Format	Description
FE 14	tail.	Subsequent call terminates current method

Description:

The **tail.** instruction must immediately precede a **call**, **calli**, or **callvirt** instruction. It indicates that the current method's stack frame should be removed before the call instruction is executed. It implies that the value returned from the following call is also the value returned by the current method, and the call can therefore be converted into a cross-method jump.

The stack must be empty except for the arguments being transferred by the following call. The instruction following the call instruction must be a **ret**. Thus the only legal code sequence is

tail.
call (or **calli** or **callvirt**) *somewhere*
ret

Correct IL must not branch to the **call** instruction, but it may branch to the subsequent **ret**.

The current frame cannot be discarded when control is transferred from untrusted code to trusted code, since this would jeopardize code identity security. Runtime security checks may therefore cause the **tail.** to be ignored, leaving a standard call instruction. Similarly, in order to allow the exit of a synchronized region to occur after the call returns, the **tail.** prefix is ignored when used to exit a method that is marked synchronized.

There may also be implementation-specific restrictions that prevent the **tail.** prefix from being obeyed in certain cases. While an implementation is free to ignore the **tail.** prefix under these circumstances, they should be clearly documented as they can affect the behavior of programs.

Rationale: *tail. calls allow some linear space algorithms to be converted to constant space algorithms and are required by some functional programming languages. In the presence of* ***ldloca*** *and* ***ldarga*** *instructions it isn't always possible for a compiler from IL to native code to optimally determine when a* ***tail.*** *can be automatically inserted.*

Exceptions:

None.

Verifiability:

Correct IL obeys the control transfer constraints listed above. In addition, no transient or managed pointers can be passed to the method being called if they point into the stack frame that is about to be removed. The return type of the method being called must be compatible with the return type of the current method. Verification requires that no transient or managed pointers are passed to the method being called, since it does not track pointers into the current frame. See the Verification Specification for more details.

unaligned. (prefix code) – subsequent pointer instruction may be unaligned

Format	Assembly Format	Description
FE 12 <**U1**>	unaligned. *alignment*	Subsequent pointer instruction may be unaligned

Stack Transition:

```
…, addr



…, addr
```

Description:

Unaligned. specifies that *address* (an unmanaged pointer, I) on the stack is not aligned to the natural size of the immediately following **ldind**, **stind, ldfld, stfld, ldobj, stobj, initblk**, or **cpblk** instruction. That is, for a **ldind.i4** instruction the alignment of *addr* may not be to a 4-byte boundary. For **initblk** and **cpblk** the default alignment is architecture dependent (4-byte on 32-bit CPUs, 8-byte on 64-bit CPUs). Code generators that do not restrict their output to a 32-bit word size (see the EE Architecture Specification and the File Format Specification) must use **unaligned.** if the alignment is not known at compile time to be 8-byte.

The value of *alignment* must be 1, 2, or 4 and means that the generated code should assume that *addr* is byte, double byte, or quad byte aligned, respectively. Note that transient pointers (type *) are always aligned.

While the alignment for a **cpblk** instruction would logically require two numbers (one for the source and one for the destination), there is no noticeable impact on performance if only the lower number is specified.

The **unaligned.** and **volatile.** prefixes may be combined in either order. They must immediately precede a **ldind**, **stind**, **ldfld, stfld, ldobj, stobj, initblk**, or **cpblk** instruction. Only the **volatile.** prefix is allowed for the **ldsfld** and **stsfld** instructions.

Exceptions:

None.

Verifiability:

Correct IL requires that **unaligned.** be immediately followed by instructions as listed above.

volatile. (prefix code) - subsequent pointer reference is volatile

Format	Assembly Format	Description
FE 13	volatile.	Subsequent pointer reference is volatile

Stack Transition:

```
..., addr



..., addr
```

Description:

volatile. specifies that *addr* is a volatile address (i.e. it may be subject to change by an external action) and the results of reading that location cannot be cached or that multiple stores to that location cannot be suppressed. Marking an access as **volatile.** affects only that single access; other accesses to the same location must be marked separately. Access to volatile locations need not be performed atomically.

The **unaligned.** and **volatile.** prefixes may be combined in either order. They must immediately precede a **ldind**, **stind**, **ldfld, stfld, ldobj, stobj, initblk**, or **cpblk** instruction. Only the **volatile.** prefix is allowed for the **ldsfld** and **stsfld** instructions.

Exceptions:

None.

Verifiability:

Correct IL requires that **volatile.** be immediately followed by instructions as listed above.

xor - bitwise XOR

Format	Assembly Format	Description
61	xor	Bitwise XOR of integer values, returns an integer

Stack Transition:

```
..., value1, value2



..., result
```

Description:

The **xor** instruction computes the bitwise XOR of the top two values on the stack and leaves the result on the stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3 Object Model Instructions

The instructions described in the base instruction set are independent of the object model being executed. Those instructions correspond closely to what would be found on a real CPU. The object model instructions are less primitive than the base instructions in the sense that they could be built out of the base instructions and calls to the underlying operating system.

Rationale: *The object model instructions provide a common, efficient implementation of a set of services used by many (but by no means all) higher-level languages. They embed in their operation a set of conventions defined by the Lightning Virtual Object System (VOS). This include (among other things):*

Field layout within an object

Layout for late bound method calls (vtables)

Memory allocation and reclamation

Exception handling

Boxing and unboxing to convert between reference-based Objects and Value Types

For more details, see the Virtual Object System Specification

box – convert value type to object reference

Format	Assembly Format	Description
78 <**T**>	box *valueType*	Convert *valueTypePtr* of type *valueType* to a true object reference

Stack Transition:

```
…, valueTypePtr



…, obj
```

Description:

Effectively a value type has two separate representations (see the EE Architecture Specification) within the runtime:

- A ‘raw’ form used when a value type is embedded within another object.
- A ‘boxed’ form, where the data in the value type is wrapped (boxed) into an object so it can exist as independent entity.

The **box** instruction converts the ‘raw’ pointer *valueTypePtr* (a managed pointer, &, a transient pointer, *, or unmanaged pointer, I), which must point to an instance of a value type, into an instance of type Object (of type **O**). This is accomplished by creating a new object instance and copying the data from *valueTypePtr* into the newly allocated object. *ValueType* is a metadata token (a **typeref** or **typedef**, see the Metadata Specification) indicating the type of *valueTypePtr*.

Exceptions:

OutOfMemoryException is thrown if there is insufficient memory to satisfy the request.

TypeLoadException is thrown if *class* cannot be found. This is typically detected when IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that *valueTypePtr* points to a location of the correct value type, and that *valueType* is a **typeref** or **typedef** metadata token for that value type.

callvirt – call a method associated, at runtime, with an object

Format	Assembly Format	Description
6F <**T**>	callvirt *method*	Call a method associated with *obj*

Stack Transition:

```
…, obj, arg1, … argN


…, returnVal (not always returned)
```

Description:

The **callvirt** instruction calls a late-bound method on an object. That is, the method is chosen based on the runtime type of *obj* rather than the compile-time class visible in the *method* metadata token. **Callvirt** can be used to call both virtual methods and interface methods. See the EE Architecture Specification for a detailed description of the IL calling sequence. The **callvirt** instruction may be immediately preceded by a **tail.** prefix to specify that the current stack frame should be released before transferring control. If the call would transfer control to a method of higher trust than the origin method the stack frame will not be released.

method is a metadata token (a **methoddef** or **methodref**, see the Metadata Specification) that provides the name, class and signature of the method to call. In more detail, **callvirt** can be thought of as follows. Associated with *obj* is the class of which it is an instance. If *obj*'s class defines a non-static method that matches the indicated method name and signature, this method is called. Otherwise all classes in the superclass chain of obj's class are checked in order. It is an error if no method is found.

Callvirt pops the object and the arguments off the evaluation stack before calling the method. If the method has a return value, it is pushed on the stack upon method completion. On the callee side, the *obj* parameter is accessed as argument 0, *arg1* as argument 1 etc.

The arguments are placed on the stack in left-to-right order. That is, the first argument is computed and placed on the stack, then the second argument, etc. The **this** pointer (always required for **callvirt**) must be pushed before any of the user-visible arguments, and it is not allowed to be null. The signature (carried in the metadata associated with *method*) need not contain an entry in the parameter list for the **this** pointer (see the Metadata Specification).

Note that a virtual method may also be called using the **call** instruction.

Exceptions:

MissingMethodException may be thrown if a non-static method with the indicated name and signature could not be found in *obj's* class or any of its superclasses. This is typically detected when IL is converted to native code, rather than at runtime.

NullReferenceException is thrown if *obj* is null.

SecurityException may be thrown if the runtime system security does not grant the caller access to the called method. The security check may occur when the IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that the destination method exists and the values on the stack correspond to the types of the parameters of the method being called.

In its typical use, **callvirt** is verifiable if (a) the above restrictions are met, (b) the verification type of *obj* is consistent with the method being called, (c) the verification types of the objects on the stack are consistent with the types expected by the method call, and (d) the method is visible from the callsite. A **callvirt** annotated by **tail.** is a special case.

castclass – cast an object to a class

Format	Assembly Format	Description
74 <T>	castclass *class*	Cast *obj* to *class*

Stack Transition:

```
…, obj



…, obj2
```

Description:

The **castclass** instruction attempts to cast *obj* (an **O**) to the *class*. *Class* is a metadata token (a **typeref** or **typedef**, see the Metadata Specification) indicating the desired class. If the class of the object on the top of the stack does not implement *class* (if *class* is an interface) and is not a subclass of *class* (if *class* is a regular class) then an **InvalidCastException** is thrown.

If *obj* is null, **castclass** succeeds and returns null. This behavior differs from **isInst**.

Notice that the **castclass** instruction may change the representation of the object: *obj* and *obj2* need not be identical.

Exceptions:

InvalidCastException is thrown if *obj* cannot be cast to *class*.

TypeLoadException is thrown if *class* cannot be found. This is typically detected when IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that *class* is a valid **TypeRef** or **TypeDef** token, and that *obj* is always either null or a reference to an object, i.e. of type **O**. Verifiable code may require the use of **castclass** to correctly detect the type of an object, as described in the verifier specification.

cpobj - copy a value type

Format	**Assembly Format**	**Description**
70 <**T**>	cpobj *classTok*	Copy a value type from *srcValObj* to *destValObj*

Stack Transition:

…, destValObj, srcValObj

…,

Description:

The **cpobj** instruction copies the value type located at the address specified by *srcValObj* (an unmanaged pointer, **I**, a transient pointer, *****, or a managed pointer, **&**) to the address specified by *destValObj* (also a pointer). Behavior is unspecified if *srcValObj* and *dstValObj* are not pointers to instances of the class represented by *classTok* (a **typeref** or **typedef**, see the Metadata Specification), or if *classTok* does not represent a value type.

Exceptions:

InvalidAddressException may be thrown if an invalid address is detected.

Rationale: *Verified code uses this instruction only when both srcValObj and destValObj are known to contain valid addresses, so an implementation may choose to check addresses or not without affecting the security of verified code.*

Verifiability:

Correct IL ensures that *classTok* is a valid **TypeRef** or **TypeDef** token for a value type, as well as that *srcValObj* and *destValObj* are both pointers to locations of that type.

Verification requires, in addition, that *srcValObj* and *destValObj* are both either transient or managed pointers (not unmanaged pointers).

initobj - initialize a value type

Format	Assembly Format	Description
FE 15 <**T**>	initobj *classTok*	Initialize a value type

Stack Transition:

```
…, addrOfValObj
```

```
…,
```

Description:

The **initobj** instruction initializes all the fields of the object represented by the address *addrOfValObj* (of type **I, &**, or *****) to **null** or a 0 of the appropriate primitive type. After this method is called, the instance is ready for the constructor method to be called. Behavior is unspecified if either *valObj* is a not pointer to instances of the class represented by *classTok* (a **typeref** or **typedef**, see the Metadata Specification), or *classTok* does not represent a value type.

Notice that, unlike **newobj**, the constructor method is not called by **initobj**. **Initobj** is intended for initializing value types, while **newobj** is used to allocate and initialize objects.

Exceptions:

None.

Verifiability:

Correct IL ensures that *classTok* is a valid **typeref** or **typedef** token specifying a value type, and that *valObj* is a pointer to an instance of that value type.

isinst – test if an object is an instance of a class or interface, returning NULL or an instance of that class or interface

Format	Assembly Format	Description
75 <**T**>	isinst *class*	test if *obj* is an instance of *class*, returning NULL or an instance of that class or interface

Stack Transition:

```
…, obj



…, result
```

Description:

The **isinst** instruction tests whether *obj* (type **O**) is an instance of *class*. *Class* is a metadata token (a **typeref** or **typedef**, see the Metadata Specification) indicating the desired class. If the class of the object on the top of the stack implements *class* (if *class* is an interface) or is a subclass of *class* (if *class* is a regular class) then it is cast to the type *class* and the result is pushed on the stack, exactly as though **castclass** had been called. Otherwise NULL is pushed on the stack. If *obj* is NULL, **isinst** returns NULL.

Exceptions:

TypeLoadException is thrown if *class* cannot be found. This is typically detected when IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that *class* is a valid **typeref** or **typedef** token indicating a class, and that *obj* is always either null or an object reference, i.e. of type **O**.

ldelem.<type> – load an element of an array

Format	Assembly Format	Description
90	ldelem.i1	Load the element at *index* with type **I1** onto the top of the stack as an **I4**
92	ldelem.i2	Load the element at *index* with type **I2** onto the top of the stack as an **I4**
94	ldelem.i4	Load the element at *index* with type **I4** onto the top of the stack as an **I4**
96	ldelem.i8	Load the element at *index* with type **I8** onto the top of the stack as an **I8**
91	ldelem.u1	Load the element at *index* with type **U1** onto the top of the stack as an **I4**
93	ldelem.u2	Load the element at *index* with type **U2** onto the top of the stack as an **I4**
94	ldelem.u4	Load the element at *index* with type **U4** onto the top of the stack as an **I4** (alias for **ldind.i4**)
96	ldelem.u8	Load the element at *index* with type **U8** onto the top of the stack as an **I4** (alias for **ldind.i8**)
98	ldelem.r4	Load the element at *index* with type **R4** onto the top of the stack as an **F**
99	ldelem.r8	Load the element at *index* with type **R8** onto the top of the stack as an **F**
97	ldelem.i	Load the element at *index* with type **I** onto the top of the stack as an **I**
9A	ldelem.ref	Load the element at *index*, an object, onto the top of the stack as an **O**

Stack Transition:

```
…,  array, index


…, value
```

Description:

The **ldelem** instruction loads the value of the element with index *index* (of type U) in the zero-based one-dimensional array *array* and places it on the top of the stack. Arrays are objects and hence represented by a value of type **O**. The return value is indicated by the instruction.

For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a **LoadElement** method.

Note that integer value of 4 bytes or less are extended to **I4** (not **I**) when they are loaded onto the evaluation stack. Floating point values are converted to **F** type when loaded onto the evaluation stack.

Exceptions:

NullReferenceException is thrown if *array* is null.

IndexOutOfRangeException is thrown if *index* is larger than the bound of *array*.

ArrayTypeMismatchException is thrown if *array* doesn't hold elements of the required type.

Verifiability:

Correct IL code requires that *array* is either null or an array.

ldelema – load address of an element of an array

Format	Assembly Format	Description
8F <**T**>	ldelema *class*	Load the address of element at *index* onto the top of the stack

Stack Transition:

```
…, array, index


…, value
```

Description:

The **ldelema** instruction loads the address of the element with index *index* (of type U) in the zero-based one-dimensional array *array* (of element type *class)* and places it on the top of the stack. Arrays are objects and hence represented by a value of type **O**. The return value is a managed pointer (type **&**).

For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a **LoadElementAddress** method.

Exceptions:

NullReferenceException is thrown if *array* is null.

IndexOutOfRangeException is thrown if *index* is larger than the bound of *array*.

ArrayTypeMismatchException is thrown if *array* doesn't hold elements of the required type.

Verifiability:

Correct IL ensures that *class* is a **typeref** or **typedef** token to a class, and that *array* is indeed always either null or an array.

ldfld – load field of an object

Format	Assembly Format	Description
7B <**T**>	ldfld *field*	Push the value of *field* of object *obj* on the stack

Stack Transition:

```
…,  obj

…, value
```

Description:

The **ldfld** instruction pushes onto the stack the value of a field of *obj***.** *obj* must be an object (type **O**), a managed pointer (type **&**), an unmanaged pointer (type **I**), a transient pointer (type *****), or an instance of a value type. The use of an unmanaged pointer is not permitted in verified code. *field* is a metadata token (a **fieldref** or **fielddef**, see the Metadata Specification) that must refer to a field member. The return type is that associated with *field*. **ldfld** pops the object reference off the stack and pushes the value for the field in its place. The field may be either an instance field (in which case *obj* must not be null) or a static field.

The **ldfld** instruction may be preceded by either or both of the **unaligned.** and **volatile.** prefixes.

Exceptions:

NullReferenceException is thrown if *obj* is null and the field is not static.

MissingFieldException is thrown if *field* is not found in the metadata. This is typically checked when IL is converted to native code, not at runtime.

Verifiability:

Correct IL ensures that *field* is a valid token referring to a non-static field, and that *obj* will always have a type compatible with that required for the lookup being performed. For verifiable code, *obj* may not be an unmanaged pointer.

ldflda – load field address

Format	Assembly Format	Description
7C <**T**>	ldflda *field*	Push the address of *field* of object *obj* on the stack

Stack Transition:

```
…,  obj



…, address
```

Description:

The **ldflda** instruction pushes the address of a field *obj*. *obj* is either an object, type **O**, a managed pointer, type **&**, an unmanaged pointer, type **I**, or a transient pointer, type *****. The use of an unmanaged pointer is not allowed in verified code. The value returned by **ldflda** is a managed pointer (type **&**) unless *obj* is an unmanaged pointer, in which case it is an unmanaged pointer (type **I**).

field is a metadata token (a **fieldref** or **fielddef**, see the Metadata Specification) that must refer to a field member.

Exceptions:

NullReferenceException is thrown if *obj* is null and the field isn't static.

MissingFieldException is thrown if *field* is not found in the metadata. This is typically checked when IL is converted to native code, not at runtime.

Verifiability:

Correct IL ensures that *field* is a valid **fieldref** token and that *obj* will always have a type compatible with that required for the lookup being performed.

Note: Using **ldflda** to compute the address of a static, init-only field and then using the resulting pointer to modify that value outside the body of the class initializer may lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by the verifier.

ldlen – load the length of an array

Format	Assembly Format	Description
8E	ldlen	push the length (of type U) of *array* on the stack

Stack Transition:

```
…, array
```

```
…, length
```

Description:

The **ldlen** instruction pushes the length of *array* (a zero-based, one-dimensional array) on the stack.

Arrays are objects and hence represented by a value of type **O**. The return value is a **U**.

Exceptions:

NullReferenceException is thrown if *array* is null.

Verifiability:

Correct IL ensures that *array* is indeed always either null or a zero-based, one dimensional array.

ldobj - copy value type to the stack

Format	Assembly Format	Description
71 <T>	ldobj *classTok*	Copy instance of value type *classTok* to the stack.

Stack Transition:

```
…, addrOfValObj



…, valObj
```

Description:

The **ldobj** instruction copies the value pointed to by *addrOfValObj* (of type managed pointer, **&**, transient pointer, *****, or unmanaged pointer, **U**) to the top of the stack. The number of bytes copied depends on the size of the class represented by *classTok*. *ClassTok* is a metadata token (a **typeref** or **typedef**, see the Metadata Specification) representing a value type.

Rationale: *The **ldobj** instruction is used to pass a value type as a parameter. See the EE Architecture Specification.*

It is unspecified what happens if *valObj* is not an instance of the class represented by *ClassTok* or if *ClassTok* does not represent a value type.

The operation of the **ldobj** instruction may be altered by an immediately preceding **volatile.** or **unaligned.** prefix instruction.

Exceptions:

TypeLoadException is thrown if *class* cannot be found. This is typically detected when IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that *classTok* is a metadata token representing a value type and that *valObj* is a pointer to a location containing an initialized value of the type specified by *classTok*. Verifiable code additionally requires that *valObj* is a transient or managed pointer.

ldsfld – load static field of a class

Format	Assembly Format	Description
7E <**T**>	ldsfld *field*	Push the value of *field* on the stack

Stack Transition:

…,

…, value

Description:

The **ldsfld** instruction pushes the value of a static (shared among all instances of a class) field on the stack. *field* is a metadata token (a **fieldref** or **fielddef**, see the Metadata Specification) referring to a static field member. The return type is that associated with *field*.

The **ldsfld** instruction may have a **volatile.** prefix.

Exceptions:

Verifiability:

Correct IL ensures that *field* is a valid metadata token referring to a static field member.

ldsflda – load static field address

Format	Assembly Format	Description
7F <**T**>	ldsflda *field*	Push the address of the static field, *field,* on the stack

Stack Transition:

…,

…, address

Description:

The **ldsflda** instruction pushes the address (a transient pointer, type *****, if *field* refers to a type whose memory is managed; otherwise an unmanaged pointer, type **I**) of a static field on the stack. *field* is a metadata token (a **fieldref** or **fielddef**, see the Metadata Specification) referring to a static field member.

Exceptions:

MissingFieldException is thrown if *field* is not found in the metadata. This is typically checked when IL is converted to native code, not at runtime.

Verifiability:

Correct IL ensures that *field* is a valid metadata token referring to a static field member.

ldstr – load a literal string

Format	Assembly Format	Description
72 <**T**>	ldstr *string*	push a string object for the literal *string*

Stack Transition:

…,

…, string

Description:

The **ldstr** instruction pushes a new string object representing the literal stored in the metadata as *string* (which must be a string literal, see the Metadata Specification).

The **ldstr** instruction allocates memory and performs any format conversion required to convert from the form used in the file to the string format required at runtime. The EE guarantees that the result of a **ldstr** instruction referring to two metadata tokens that have the same sequence of characters returns precisely the same string object (a process known as "string interning").

Exceptions:

None.

Verifiability:

Correct IL requires that *mdToken* is a valid string literal metadata token.

ldtoken - load the runtime representation of a metadata token

Format	Assembly Format	Description
D0 <**T**>	ldtoken *token*	Convert metadata *token* to its runtime representation

Stack Transition:

…

…, `unmanaged pointer`

Description:

The **ldtoken** instruction pushes an unmanaged pointer (type **I**) to the runtime representation of a metadata token. The token must be one of:

A **methoddef** or **methodref** describing a particular method, in which case the description of the method (not the address, cf. **ldftn** and **ldvirtftn**) is pushed on the stack.

A **typedef** or **typeref** describing a class, value type, or implementation.

A **fielddef** or **fieldref** describing a particular field of a class, value type, or implementation.

The value pushed on the stack is useful only for special-purpose class library routines such as those that are used for type-safe access to a variable argument list (see the **arglist** instruction).

Exceptions:

None.

Verifiability:

Correct IL requires that *token* describes a valid metadata token.

ldvirtftn - load a virtual method pointer

Format	Assembly Format	Description
FE 07 <**T**>	ldvirtftn *mthd*	Push address of virtual method *mthd* on the stack

Stack Transition:

```
… object

…, ftn
```

Description:

The **ldvirtftn** instruction pushes an unmanaged pointer (type **I**) to the native code implementing the virtual method associated with *object* and described by the method reference *mthd* (a metadata token, either a **methoddef** or **methodref**, see the Metadata Specification) onto the stack. The value pushed can be called using the **calli** instruction if it references a managed method (or a stub that transitions from managed to unmanaged code).

The value returned points to native code using the calling convention specified by *method*. Thus a method pointer can be passed to unmanaged native code (e.g. as a callback routine) if that routine expects the corresponding calling convention. Note that the address computed by this instruction may be to a thunk produced specially for this purpose (for example, to re-enter the EE when a native version of the method isn't available).

Exceptions:

None.

Verifiability:

Correct IL ensures that *method* is a valid **methoddef** or **methodref** token. The verifier tracks the type of the value pushed in more detail than the "**I**" type, remembering that it is a method pointer. Such a method pointer can then be used in verified code with **calli** or to construct a delegate.

mkrefany – push a typed reference on the stack

Format	Assembly Format	Description
C6 <**T**>	mkrefany *class*	push a typed reference to *ptr* of type *class* onto the stack

Stack Transition:

```
…, ptr


…, typedRef
```

Description:

The **mkrefany** instruction supports the passing of dynamically typed references. *Ptr* must be a pointer (type **&,** *****, or **I**) which is the address of a piece of data. *Class* is the class token (a **typeref** or **typedef**, see the Metadata Specification) describing the type of *ptr*. **Mkrefany** pushes a typed reference on the stack, an opaque descriptor of *ptr* and *class*. The only legal operation on a typed reference on the stack is to pass it to a method that requires a typed reference as a parameter. The callee can then use the **refanytype** and **refanyval** instructions to retrieve the type (*class*) and address (*ptr*) respectively.

The verifier will fail if it cannot deduce that *ptr* is a pointer to an instance of *class*.

Exceptions:

TypeLoadException is thrown if *class* cannot be found. This is typically detected when IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that *class* is a valid **typeref** or **typedef** token describing some type and that *ptr* is pointer to that type. Verification additionally requires that *ptr* be a transient or managed pointer.

newarr – create a zero-based, one-dimensional array

Format	Assembly Format	Description
8D <**T**>	newarr *etype*	create a new array with elements of type *etype*

Stack Transition:

…, *numElems*

…, *array*

Description:

The **newarr** instruction pushes a reference to a new zero-based, one-dimensional array whose elements are of type *elemtype***,** a metadata token (a **typeref** or **typedef**, see the Metadata Specification). *Numelems* (of type U) indicates the array bound (note: this is an *un*signed integer). Valid array indexes are 0 ≤ index < *numElems*. The elements of an array can be any type, including value types.

Zero-based, one-dimensional arrays of numbers are created using a metadata token referencing the appropriate value type (**System.Int32**, etc.). Elements of the numeric arrays are initialized to 0 of the appropriate type.

One-dimensional arrays that aren't zero-based and multidimensional arrays are created using **newobj** rather than **newarr**. More commonly, they are created using the methods of System.Array class in the NGWS Base Framework.

Exceptions:

OutOfMemoryException is thrown if there is insufficient memory to satisfy the request.

Verifiability:

Correct IL ensures that *elemType* is a valid **typeref** or **typedef** token.

newobj – create a new object

Format	Assembly Format	Description
73 <T>	newobj *ctor*	allocate an uninitialized object and call *ctor*

Stack Transition:

…, *arg1*, … *argN*

…, *obj*

Description:

The **newobj** instruction creates a new object or a new instance of a value type. *Ctor* is a metadata token (a **methodref** or **methodef** that must be marked as a constructor, see the Metadata Specification) that indicates the name, class and signature of the constructor to call. If a constructor exactly matching the indicated name, class and signature cannot be found, **MissingMethodException** is thrown.

The **newobj** instruction allocates a new instance of the class associated with *constructor* and initializes all the fields in the new instance to 0 (of the proper type) or **null** as appropriate. It then calls the constructor with the given arguments along with the newly created instance. After the constructor has been called, the now initialized object reference is pushed on the stack.

From the constructor's point of view, the uninitialized object is argument 0 and the other arguments passed to **newobj** follow in order.

All zero-based, one-dimensional arrays are created using **newarr**, not **newobj**. On the other hand, all other arrays (more than one dimension, or one-dimensional but not zero-based) are created using **newobj**.

Value types are not usually created using **newobj**. They are usually allocated either as arguments or local variables, using **newarr** (for zero-based, one-dimensional arrays), or as fields of objects. Once allocated, they are initialized using **initobj**. However, the **newobj** instruction can be used to create a new instance of a value type on the stack, which can then be passed as an argument, stored in a local, etc.

Exceptions:

OutOfMemoryException is thrown if there is insufficient memory to satisfy the request.

MissingMethodException is thrown if a constructor method with the indicated name, class and signature could not be found. This is typically detected when IL is converted to native code, rather than at runtime.

Verifiability:

Correct IL ensures that *constructor* is a valid **methodref** or **methoddef** token, and that the arguments on the stack are compatible with those expected by the constructor. The verifier considers a delegate constructor as a special case, checking that the method pointer passed in as the second argument under the notional type **I** does indeed refer to a method of the correct type.

refanytype – load the type out of a typed reference

Format	Assembly Format	Description
FE 1D	refanytype	Push the type token stored in a typed reference

Stack Transition:

```
…,  TypedRef


…, type
```

Description:

Retrieves the type token embedded in *TypedRef*. See the **mkrefany** instruction.

Exceptions:

None.

Verifiability:

The **refanytype** instruction is always verifiable.

refanyval – load the address out of a typed reference

Format	Assembly Format	Description
C2 <**T**>	refanyval *type*	Push the address stored in a typed reference

Stack Transition:

```
…,  TypedRef



…, address
```

Description:

Retrieves the address (of type &) embedded in *TypedRef*. The type of reference in *TypedRef* must match the type specified by *type* (a metadata token, either a **typedef** or a **typeref**, see the Metadata Specification). See the **mkrefany** instruction.

Exceptions:

None.

Verifiability:

The **refanyval** instruction is always verifiable.

rethrow – rethrow the current exception

Format	Assembly Format	Description
FE 1A	rethrow	Rethrow the current exception

Stack Transition:

…,

…,

Description:

The **rethrow** instruction is only permitted within the body of a **catch** handler (see the Exception Specification). It throws the same exception that was caught by this handler.

Exceptions:

The original exception is thrown.

Verifiability:

Correct IL uses this instruction only within the body of a **catch** handler.

sizeof – load the size in bytes of a value type

Format	**Assembly Format**	**Description**
FE 1C <**T**>	sizeof *valueType*	Push the size, in bytes, of a value type as a **U4**

Stack Transition:

```
…,



…, size (4 bytes, unsigned)
```

Description:

Returns the size, in bytes, of a value type. *ValueType* must be a metadata token (a **typeref** or **typedef**, see the Metadata Specification) the specifies a value type.

Rationale: *The definition of a value type can change between the time the IL is generated and the time that it is loaded for execution. Thus, the size of the type is not always known when the IL is generated. The* ***sizeof*** *instruction allows IL code to determine the size at runtime without the need to call into the base class library. The computation can occur entirely at JIT time.*

Exceptions:

None.

Verifiability:

Correct IL ensures that *valueType* is a **typeref** or **typedef** referring to a value type.

stelem.<type> – store an element of an array

Format	Assembly Format	Description
9C	stelem.i1	Replace array element at *index* with the **I1** value on the stack
9D	stelem.i2	Replace array element at *index* with the **I2** value on the stack
9E	stelem.i4	Replace array element at *index* with the **I4** value on the stack
9F	stelem.i8	Replace array element at *index* with the **I8** value on the stack
A0	stelem.r4	Replace array element at *index* with the **R4** value on the stack
A1	stelem.r8	Replace array element at *index* with the **R8** value on the stack
9B	stelem.i	Replace array element at *index* with the **i** value on the stack
A2	stelem.ref	Replace array element at *index* with the **ref** value on the stack

Stack Transition:

```
…, array, index, value



…,
```

Description:

The **stelem** instruction replaces the value of the element with zero-based index *index* (of type U) in the one-dimensional array *array* with *value*. Arrays are objects and hence represented by a value of type **O**.

Note that **stelem.ref** implicitly casts *value* to the element type of *array* before assigning the value to the array element. This cast can fail, even for verified code. Thus the **stelem.ref** instruction may throw the **InvalidCastException**.

For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a **StoreElement** method.

Exceptions:

NullReferenceException is thrown if *array* is null.

IndexOutOfRangeException is thrown if *index* is larger than the bound of *array*.

ArrayTypeMismatchException is thrown if *array* doesn't hold elements of the required type.

Verifiability:

Correct IL requires that *array* be a zero-based, one-dimensional array.

stfld – store into a field of an object

Format	Assembly Format	Description
7D <**T**>	stfld *field*	Replace the value of *field* of the object *obj* with *val*

Stack Transition:

```
…, obj, value
```

…,

Description:

The **stfld** instruction replaces the value of a field of an *obj* (an **O**) or via a pointer (type **I**, **&**, or *****) with *value*. *field* is a metadata token (a **fieldref** or **fielddef**, see the Metadata Specification) that refers to a field member reference. **stfld** pops the value and the object reference off the stack and updates the object.

The **stfld** instruction may have a prefix of either or both of **unaligned.** and **volatile.**.

Exceptions:

NullReferenceException is thrown if *obj* is null and the field isn't static.

MissingFieldException is thrown if *field* is not found in the metadata. This is typically checked when IL is converted to native code, not at runtime.

Verifiability:

Correct IL ensures that *field* is a valid token referring to a non-static field, and that *obj* and *value* will always have types appropriate for the assignment being performed. For verifiable code, *obj* may not be an unmanaged pointer.

Note: Using **stfld** to change the value of a static, init-only field outside the body of the class initializer may lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by the verifier.

stobj - store a value type from the stack into memory

Format	Assembly Format	Description
81 <**T**>	stobj *classTok*	Store a value of type *classTok* from the stack into memory

Stack Transition:

`…, addr, valObj`

`…,`

Description:

The **stobj** instruction copies the value type *valObj* into the address specified by *addr* (a pointer of type **I**, *****, or **&**). The number of bytes copied depends on the size of the class represented by *classTok*. *ClassTok* is a metadata token (a **typeref** or **typedef**, see the Metadata Specification) representing a value type.

It is unspecified what happens if *valObj* is not an instance of the class represented by *ClassTok* or if *classTok* does not represent a value type.

The operation of the **stobj** instruction may be altered by an immediately preceding **volatile.** or **unaligned.** prefix instruction.

Exceptions:

TypeLoadException is thrown if *class* cannot be found. This is typically detected when IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that *classTok* is a metadata token representing a value type and that *valObj* is a pointer to a location containing an initialized value of the type specified by *classTok*. In addition, verifiable code requires that *valObj* be a transient or managed pointer.

stsfld – store a static field of a class

Format	Assembly Format	Description
80 <T>	stsfld *field*	Replace the value of *field* with *val*

Stack Transition:

…, *val*

…,

Description:

The **stsfld** instruction replaces the value of a static field with a value from the stack. *field* is a metadata token (a **fieldref** or **fielddef**, see the Metadata Specification) that must refer to a static field member. **Stsfld** pops the value off the stack and updates the static field.

The **stsfld** instruction may be prefixed by **volatile.**.

Exceptions:

MissingFieldException is thrown if *field* is not found in the metadata. This is typically checked when IL is converted to native code, not at runtime.

Verifiability:

Correct IL ensures that *field* is a valid token referring to a static field, and that *value* will always have a type appropriate for the assignment being performed.

throw – throw an exception

Format	Assembly Format	Description
7A	throw	Throw an exception

Stack Transition:

```
…, object
```

```
…,
```

Description:

The **throw** instruction throws the exception *object* (type **O**) on the stack. For details of the exception mechanism, see the Exception Specification .

While the EE permits any object to be thrown, the common language subset (CLS) describes a specific exception class that must be used for language interoperability.

Exceptions:

NullReferenceException is thrown if *obj* is null.

Verifiability:

Correct IL ensures that *class* a valid **TypeRef** token indicating a class, and that *obj* is always either null or an object reference, i.e. of type **O**.

unbox – Convert boxed value type to its raw form

Format	Assembly Format	Description
79 <**T**>	unbox *valuetype*	Extract the value type data from *obj*, its boxed representation

Stack Transition:

```
…, obj
```

```
…, valueTypePtr
```

Description:

Effectively a value type has two separate representations (see the EE Architecture Specification) within the runtime:

- A 'raw' form used when a value type is embedded within another object.
- A 'boxed' form, where the data in the value type is wrapped (boxed) into an object so it can exist as an independent entity.

The **unbox** instruction converts *obj* (of type **O**), the boxed representation of a value type, to *valueTypePtr* (a managed pointer, type **&**), its unboxed form. *ValueType* is a metadata token (a **typeref** or **typedef**, see the Metadata Specification) indicating the type of value type contained within *obj*. If *obj* is not a boxed instance of *ValueType*, an **InvalidCastException** is thrown.

Unlike **box**, which is required to make a copy of a value type for use in the object, **unbox** is *not* required to copy the value type from the object. Typically it simply computes the address of the value type that is already present inside of the boxed object.

Exceptions:

InvalidCastException is thrown if *obj* is not a boxed *valueType*.

NullReferenceException is thrown if obj is null.

TypeLoadException is thrown if *class* cannot be found. This is typically detected when IL is converted to native code rather than at runtime.

Verifiability:

Correct IL ensures that *valueType* is a **typeref** or **typedef** metadata token for some value type, and that *obj* is always an object reference, i.e. of type **O**.

4 Annotations

In addition to the instructions described earlier, IL includes a set of instructions that can be ignored by most processors. They are used to convey additional information for the use of particular IL processors (typically for use in OptIL, a specialized version of IL created by an optimizing compiler and consumed by a special JIT compiler). See the separate OptIL specification.

ann.call – start of simple calling sequence

Format	Assembly Format	Description
C7 <**T**>	ann.call *signature*	Begin the calling sequence for a method of the specified *signature*

Stack Transition:

…,

…,

Description:

The **ann.call** instruction flags the start of a simple calling sequence that will be terminated by a call instruction (**call**, **calli**, **callvirt**, **jmp**, or **jmpi**) and will pass arguments as specified by *signature*, a metadata token (a stand-alone signature, see the Metadata Specification). As with the **calli** instruction, *signature* specifies the number and type of the arguments being passed as well as the calling convention.

There are significant restrictions on the code that is permitted to occur between the **ann.call** instruction and the call instruction with which it corresponds. These are documented in the OptIL specification.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.catch – start an exception filter or handler

Format	Assembly Format	Description
C8	ann.catch	start an exception filter or handler

Stack Transition:

…,

…,

Description:

The **ann.catch** instruction indicates that an exception filter or handler is beginning. At these locations the stack contains an item that cannot be predicted by a simple scan of the IL instruction stream.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.data – multi-byte no operation

Format	**Assembly Format**	**Description**
FE 22 <**U4**> …	ann.data *count* …	multi-byte no operation

Stack Transition:

…,

…,

Description:

The **ann.data** instruction allows uninterpreted information to be inserted in the instruction stream. There are *count* bytes of data following the 4-byte *count* in the instruction.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.dead – stack location is no longer live

Format	Assembly Format	Description
C9 <**U2**>	ann.dead *location*	stack location is no longer live

Stack Transition:

…,

…,

Description:

The **ann.dead** instruction notifies an IL processor that a stack location (local variable or argument) that would otherwise appear to contain a legitimate value should not, in fact, be reported to the garbage collector. If *location* is 0 or greater, the *location*th local variable is now dead. If *location* is negative, the *location*th argument (numbered from –1 as leftmost argument) is now dead. See the OptIL specification for details of the OptJIT liveness tracking algorithm.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.def – SSA definition node

Format	Assembly Format	Description
CD	ann.def	SSA definition node

Stack Transition:

…,

…,

Description:

The **ann.def** instruction is used to embed an SSA (single static assignment) graph into the IL instruction stream. The **ann.def** instruction assigns a node number to the output of the next IL instruction. The node numbers are assigned sequentially, from 0, through the method. A new node number is allocated for each **ann.def** and **ann.phi** instruction.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.hoisted– start of the simple portion of a hoisted calling sequence

Format	Assembly Format	Description
CA	ann.hoisted	start of the simple portion of a hoisted calling sequence

Stack Transition:

…,

…,

Description:

The **ann.hoisted** instruction must follow an **ann.hoisted_call** instruction. It indicates that the complex portion of the argument evaluation has been completed and that the subsequent instructions are part of a simple calling sequence. The overall calling sequence begins with the **ann.hoisted** instruction and terminates with the subsequent call instruction (**call**, **calli**, **callvirt**, **jmp**, or **jmpi**). See the OptIL specification. for details.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.hoisted_call – start of complex argument evaluation

Format	**Assembly Format**	**Description**
CB <**T**>	ann.hoisted_call *signature*	start of argument evaluation for a call to a method with the specified *signature*

Stack Transition:

…,

…,

Description:

The **ann.hoisted_call** instruction flags the start of a calling sequence that will be terminated by a call instruction (**call**, **calli**, **callvirt**, **jmp**, or **jmpi**) and will pass arguments as specified by *signature*, a metadata token (a stand-alone signature, see the Metadata Specification). As with the **calli** instruction, *signature* specifies the number and type of the arguments being passed as well as the calling convention.

Unlike calls that use the **ann.call** instruction the arguments to be passed with **ann.hoisted_call** can be arbitrarily complex, but the calling sequence is divided into two parts. The complex evaluation is performed starting with the **ann.hoisted_call** instruction, then argument computation, then an **ann.hoisted** instruction, then a simple calling sequence. The details are documented in the OptIL specification.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.live – mark a stack location as live

Format	**Assembly Format**	**Description**
FE 16 <**U2**>	ann.live *location*	Mark a stack location as live

Stack Transition:

…,

…,

Description:

The **ann.live** instruction notifies an IL processor that a stack location (local variable or argument) that would otherwise appear not to contain a legitimate value should, in fact, be reported to the garbage collector. If *location* is 0 or greater, the *location*th local variable is now live. If *location* is negative, the *location*th argument (numbered from –1 as leftmost argument) is now live. See the OptIL specification for details of the OptJIT liveness tracking algorithm.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.phi – SSA Φ node

Format	**Assembly Format**	**Description**
CF <**U1**> <**U2**> …	ann.def *n* *node₁* …	SSA definition node

Stack Transition:

…,

…,

Description:

The **ann.phi** instruction is used to embed an SSA (single static assignment) graph into the IL instruction stream. The **ann.phi** instruction indicates that *n* existing nodes ($node_i$) are to be merged into a new node. Node numbers are assigned sequentially, from 0, through the method. A new node number is allocated for each **ann.def** and **ann.phi** instruction.

Exceptions:

None.

Verifiability:

Not verifiable.

ann.ref.<length> – SSA reference node

Format	**Assembly Format**	**Description**
FE 19 <**U2**>	ann.ref *n*	SSA definition node
CE <**U1**>	ann.ref.s *n*	SSA definition node, short form

Stack Transition:

…,

…,

Description:

The **ann.ref** instruction is used to embed an SSA (single static assignment) graph into the IL instruction stream. The **ann.ref** instruction specifies that the output of the next IL instruction is the same as the value computed at node *n*. The node numbers are assigned sequentially, from 0, through the method. A new node number is allocated for each **ann.def** and **ann.phi** instruction.

Exceptions:

None.

Verifiability:

Not verifiable.

5 Sample Code Sequences

There should be sample on delegates, value types, ref-any, and varargs at the very least.

5.1 Value types

To be supplied. Fragments:

For example, to create a value type **MyValueType** in the third local variable, use the following code sequence:

```
ldloca 3                 ; Load address of variable
dupRef                   ; For constructor
initobj MyValueType      ; Clear the instance
call MyValueType::<init> ; Call the constructor
```

For example, to pass local variable 3 as a parameter to a method, the following code is generate:

```
ldloca 3 ; Address of local variable 3
ldobj    ; Copy to stack
```

For example, if the second argument to a method is a value type named **MyValueType** that contains a field named **MyField**, the contents of that field can be accessed as follows:

```
ldarga 2 ; Address of argument 2
ldfld MyValueType::MyField
```

For example consider the body of an instance method of a value type that returns a value type as a result. This will have two hidden parameters: argument 0 is the **this** pointer (a by-ref pointer to the method’s instance) and argument 1 is the address where the return value should be stored. So to return the instance itself as the result, the following code is generated:

```
ldarga 0 ; Address of instance itself
ldarga 1 ; Address for returned value
cpobj    ; Copy this to return value
```