Common Language Infrastructure (CLI)

Partition III:
CIL Instruction Set

Final draft – October 2002

Produced by ECMA TC39/TG3

Table of Contents

1 Scope

This specification is a detailed description of the Common Intermediate Language (CIL) instruction set, part of the specification of the Common Language Infrastructure. Partition I describes the architecture of the CLI and provides an overview of a large number of issues relating to the CIL instruction set. That overview is essential to an understanding of the instruction set as described here.

Each instruction description describes a set of related CLI machine instructions. Each instruction definition consists of five parts:

- A table describing the binary format, assembly language notation and description of each variant of the instruction. See Section 1.2 .
- A stack transition diagram that describes the state of the evaluation stack before and after the instruction is executed. See Section 1.3.
- An English description of the instruction. See Section 1.4.
- A list of exceptions that might be thrown by the instruction. See Partition I for details. There are three exceptions which may be thrown by any instruction and are not listed with the instruction:

  `ExecutionEngineException` indicates that the internal state of the Execution Engine is corrupted and execution cannot continue. [**Note:** in a system that executes only verifiable code this exception is not thrown.]

  `StackOverflowException` indicates that the hardware stack size has been exceeded. The precise timing of this exception and the conditions under which it occurs are implementation specific. [**Note:** this exception is unrelated to the maximum stack size described in clause 1.7.4. That size relates to the depth of the evaluation stack that is part of the method state described in Partition I, while this exception has to do with the implementation of that method state on physical hardware.]

  `OutOfMemoryException` indicates that the available memory space has been exhausted, either because the instruction inherently allocates memory (`newobj`, `newarr`) or for an implementation-specific reason (for example, an implementation based on just-in-time compilation to native code may run out of space to store the translated method while executing the first `call` or `callvirt` to a given method).
- A section describing the verifiability conditions associated with the instruction. See Section 1.8.

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 Section 1.5.

Note that not all instructions are included in all CLI Profiles. See Partition IV for details.

1.1 Data Types

While the Common Type System (CTS) defines a rich type system and the Common Language Specification (CLS) specifies a subset that can be used for language interoperability, the CLI itself deals with a much simpler set of types. These types include user-defined value types and a subset of the built-in types. The subset is collectively known as the “basic CLI types”:

- A subset of the full numeric types (**`int32`**, **`int64`**, **`native int`**, and **`F`**)
- Object references (**`O`**) without distinction between the type of object referenced
- Pointer types (**`native unsigned int`** and **`&`**) without distinction as to the type pointed to

Note that object references and pointer types may be assigned the value **`null`**. This is defined throughout the CLI to be zero (a bit pattern of all bits zero)

1.1.1 Numeric Data Types

- The CLI only operates on the numeric types `int32` (4-byte signed integers), `int64` (8-byte signed integers), `native int` (native size integers), and `F` (native size floating-point numbers). The CIL instruction set, however, allows additional data types to be implemented:
- **Short integers**. 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.

Note: Short integers are loaded as 4-byte numbers on all architectures and these 4-byte numbers must always be tracked as distinct from 8-byte numbers. This helps portability of code by ensuring that the default arithmetic behavior (i.e when no `conv` or `conv.ovf` instruction are executed) will have identical results on all implementations.

Convert instructions that yield short integer values actually leave an `int32` (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 CLI 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 is modeled as storing into a short integer within the called procedure (i.e. the CLI automatically truncates) and then loading from a short integer within the calling procedure (i.e. the CLI 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 CIL instruction sequence is unaffected and it is as though the CIL 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 the excess significant bits should be ignored; 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 may be expensive, whereas 32-bit arithmetic is available and efficient on current 64-bit hardware. For this reason, numeric instructions allow `int32` and `I` data types to be intermixed (yielding the largest type used as input), but these

types *cannot* be combined with `int64`s. Instead, a `native int` or `int32` must be explicitly converted to `int64` before it can be combined with an `int64`.

- **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**. See also Partition I, Handling of Floating Point Datatypes. Storage locations for floating-point numbers (statics, array elements, and fields of classes) are of fixed size. The supported storage sizes are `float32` and `float64`. Everywhere else (on the evaluation stack, as arguments, as return types, and as local variables) floating-point numbers are represented using an internal floating-point type. In each such instance, the nominal type of the variable or expression is either `float32` or `float64`, but its value may be represented internally with additional range and/or precision. The size of the internal floating-point representation is implementation-dependent, may vary, and shall have precision at least as great as that of the variable or expression being represented. An implicit widening conversion to the internal representation from `float32` or `float64` is performed when those types are loaded from storage. The internal representation is typically the natural size for the hardware, or as required for efficient implementation of an operation. The internal representation shall have the following characteristics:
  - The internal representation shall have precision and range greater than or equal to the nominal type.
  - Conversions to and from the internal representation shall preserve value. [Note: This implies that an implicit widening conversion from `float32` (or `float64`) to the internal representation, followed by an explicit conversion from the internal representation to `float32` (or `float64`), will result in a value that is identical to the original `float32` (or `float64`) value.]

Note: The above specification allows a compliant implementation to avoid rounding to the precision of the target type on intermediate computations, and thus permits the use of wider precision hardware registers, as well as the application of optimizing transformations which result in the same or greater precision, such as contractions. Where exactly reproducible behavior is required by a language or application, explicit conversions may be used.

When a floating-point value whose internal representation has greater range and/or precision than its nominal type is put in a storage location, it is automatically coerced to the type of the storage location. This may involve a loss of precision or the creation of an out-of-range value (NaN, +infinity, or -infinity). However, the value may be retained in the internal representation for future use, if it is reloaded from the storage location without having been modified. It is the responsibility of the compiler to ensure that the memory location is still valid at the time of a subsequent load, taking into account the effects of aliasing and other execution threads (see memory model section). This freedom to carry extra precision is not permitted, however, following the execution of an explicit conversion (`conv.r4` or `conv.r8`), at which time the internal representation must be exactly representable in the associated type.

Note: To detect values that cannot be converted to a particular storage type, use a conversion instruction (`conv.r4`, or `conv.r8`) and then check for an out-of-range value using `ckfinite`. To detect underflow when converting to a particular storage type, a comparison to zero is required before and after the conversion.

Note: This standard does not specify the behavior of arithmetic operations on denormalized floating point numbers, nor does it specify when or whether such representations should be created. This is in keeping with IEC 60559:1989. In addition, this standard does not specify how to access the exact bit pattern of NaNs that are created, nor the behavior when converting a NaN between 32-bit and 64-bit representation. All of this behavior is deliberately left implementation-specific.

1.1.2 Boolean Data Type

A CLI Boolean type occupies one byte in memory. A bit pattern of all zeroes denotes a value of false. A bit pattern with any one or more bits set (analogous to a non-zero integer) denotes a value of true.

1.1.3 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 CIL 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.

1.1.4 Runtime Pointer Types

There are two kinds of pointers: unmanaged pointers and managed pointers. For pointers into the same array or object (see Partition I), 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 must 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.

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.

1.1.4.1 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 CLI), 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 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.

- Unmanaged pointers should be treated as unsigned (i.e. use **`conv.ovf.u`** rather than **`conv.ovf.i`**, etc.).
- Verifiable code cannot use unmanaged pointers to reference memory.
- 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:
  a. The unmanaged pointer refers to memory that is not in memory managed by the garbage collector
  b. The unmanaged pointer refers to a field within an object

c. The unmanaged pointer refers to an element within an array

d. The unmanaged pointer refers to the location where the element following the last element in an array would be located

1.1.4.2 Managed Pointers (type &)

Managed pointers (`&`) may point to a local variable, a method argument, 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`. (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.

- 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 CLI. This conversion is safe if any of the following is known to be true:

  a. the managed pointer does not point into the garbage collector's memory area

  b. the memory referred to has been pinned for the entire time that the unmanaged pointer is in use

  c. a garbage collection cannot occur while the unmanaged pointer is in use

  d. the garbage collector for the given implementation of the CLI is known to not move the referenced memory

1.2 Instruction Variant Table

In Chapter 3 an Instruction Variant Table is presented for each instruction. It describes each variant of the instructions. The "Format" column of the table lists the opcode for the instruction variant, along with any arguments that follow the instruction in the instruction stream. For example:

Format	**Assembly Format**	**Description**
FE 0A <*unsigned int16*>	Ldarga *argNum*	fetch the address of argument *argNum*.
0F <*unsigned int8*>	Ldarga.s *argNum*	fetch the address of argument *argNum*, short form

The first one or two hex numbers in the "Format" column show how this instruction is encoded (its "opcode"). So, the `ldarga` instruction is encoded as a byte holding FE, followed by another holding 0A. Italicized type names represent numbers that should follow in the instruction stream. In this example a 2-byte quantity that is to be treated as an unsigned integer directly follows the FE 0A opcode.

Any of the fixed size built-in types (`int8`, `unsigned int8`, `int16`, `unsigned int16`, `int32`, `unsigned int32`, `int64`, `unsigned in64`, `float32`, and `float64`) can appear in 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 4-byte integers. All argument numbers are encoded least-significant-byte-at-smallest-address (a pattern commonly termed "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 for each 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

CIL opcodes are one or more bytes long; they may be followed by zero or more operand bytes. All opcodes whose first byte lies in the ranges 0x00 through 0xEF, or 0xFC through 0xFF are reserved for standardization. Opcodes whose first byte lies in the range 0xF0 through 0xFB inclusive, are available for experimental purposes. The use of experimental opcodes in any method renders the method invalid and hence unverifiable.

The currently defined encodings are specified in Table 1: Opcode Encodings.

Table 1: Opcode Encodings

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
0x4b	ldind.u4
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
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
0x8c	box
0x8d	newarr
0x8e	ldlen
0x8f	ldelema
0x90	ldelem.i1
0x91	ldelem.u1
0x92	ldelem.i2
0x93	ldelem.u2
0x94	ldelem.i4
0x95	ldelem.u4
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
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 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 0x17	cpblk
0xfe 0x18	initblk
0xfe 0x1a	rethrow
0xfe 0x1c	sizeof
0xfe 0x1d	refanytype

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 Operand Type Table

Many CIL operations take numeric operands on the stack. These operations fall into several categories, depending on how they deal with the types of the operands. The following tables summarize the valid kinds of operand types and the type of the result. Notice that the type referred to here is the type as tracked by the CLI rather than the more detailed types used by tools such as CIL verification. The types tracked by the CLI are: **`int32`**, **`int64`**, **`native int`**, **`F`**, **`O`**, and **`&`**.

A **`op`** B (used for **`add`**, **`div`**, **`mul`**, **`rem`**, and **`sub`**). The table below shows the result type, for each possible combination of operand types. Boxes holding simply a result type, apply to all five instructions. Boxes marked ✖ indicate an invalid CIL instruction. Shaded boxes indicate a CIL instruction that is not verifiable. Boxes with a list of instructions are valid only for those instructions.

Table 2: Binary Numeric Operations

A's Type	**B's Type**					
	int32	**int64**	**native int**	**F**	**&**	**O**
int32	int32	✖	native int	✖	& (add)	✖
int64	✖	int64	✖	✖	✖	✖
native int	native int	✖	native int	✖	& (add)	✖
F	✖	✖	✖	F	✖	✖
&	& (add, sub)	✖	& (add, sub)	✖	native int (sub)	✖
O	✖	✖	✖	✖	✖	✖

Used for the **`neg`** instruction. Boxes marked ✖ indicate an invalid CIL instruction. All valid uses of this instruction are verifiable.

Table 3: Unary Numeric Operations

Operand Type	`int32`	`int64`	`native int`	`F`	`&`	`O`
Result Type	`int32`	`int64`	`native int`	`F`	✖	✖

These return a boolean value or branch based on the top two values on the stack. Used for **`beq`**, **`beq.s`**, **`bge`**, **`bge.s`**, **`bge.un`**, **`bge.un.s`**, **`bgt`**, **`bgt.s`**, **`bgt.un`**, **`bgt.un.s`**, **`ble`**, **`ble.s`**, **`ble.un`**, **`ble.un.s`**, **`blt`**, **`blt.s`**, **`blt.un`**,

blt.un.s, **bne.un**, **bne.un.s**, **ceq**, **cgt**, **cgt.un**, **clt**, **clt.un**. Boxes marked ✓ indicate that all instructions are valid for that combination of operand types. Boxes marked ✖ indicate invalid CIL sequences. Shaded boxes boxes indicate a CIL instruction that is not verifiable. Boxes with a list of instructions are valid only for those instructions.

Table 4: Binary Comparison or Branch Operations

	int32	**int64**	**native int**	**F**	**&**	**O**
int32	✓	✖	✓	✖	✖	✖
int64	✖	✓	✖	✖	✖	✖
native int	✓	✖	✓	✖	Beq[.s], bne.un[.s], ceq	✖
F	✖	✖	✖	✓	✖	✖
&	✖	✖	beq[.s], bne.un[.s], ceq	✖	✓[1]	✖
O	✖	✖	✖	✖	✖	beq[.s], bne.un[.s], ceq[2]

1. Except for **beq**, **bne.un** (or short versions) or **ceq** these combinations make sense if both operands are known to be pointers to elements of the same array. However, there is no security issue for a CLI that does not check this constraint

Note: if the two operands are *not* pointers into the same array, then the result is simply the distance apart in the garbage-collected heap of two unrelated data items. This distance apart will almost certainly change at the next garbage collection. Essentially, the result cannot be used to compute anything useful

2. **cgt.un** is allowed and verifiable on ObjectRefs (O). This is commonly used when comparing an ObjectRef with null (there is no "compare-not-equal" instruction, which would otherwise be a more obvious solution)

These operate only on integer types. Used for **and**, **div.un**, **not**, **or**, **rem.un**, **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 CLI Specification, however, the CLI 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 native unsigned int. Boxes marked ✖ indicate invalid CIL sequences. All other boxes denote verifiable combinations of operands.

Table 5: Integer Operations

	int32	**int64**	**native int**	**F**	**&**	**O**
int32	int32	✖	native int	✖	✖	✖
int64	✖	int64	✖	✖	✖	✖
native int	native int	✖	native int	✖	✖	✖
F	✖	✖	✖	✖	✖	✖
&	✖	✖	✖	✖	✖	✖
O	✖	✖	✖	✖	✖	✖

Below are the legal combinations of operands and result for the shift instructions: **shl**, **shr**, **shr_un**. Boxes marked ✖ indicate invalid CIL sequences. All other boxes denote verifiable combinations of operand. If the

"Shift-By" operand is larger than the width of the "To-Be-Shifted" operand, then the results are implementation-defined. (eg shift an int32 integer left by 37 bits)

Table 6: Shift Operations

		Shift-By					
		int32	**int64**	**native int**	**F**	**&**	**O**
To Be Shifted	**int32**	int32	✗	int32	✗	✗	✗
	int64	int64	✗	int64	✗	✗	✗
	native int	native int	✗	native int	✗	✗	✗
	F	✗	✗	✗	✗	✗	✗
	&	✗	✗	✗	✗	✗	✗
	O	✗	✗	✗	✗	✗	✗

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 boxes marked ✗ indicate invalid CIL sequences.

Table 7: Overflow Arithmetic Operations

	int32	**int64**	**native int**	**F**	**&**	**O**
int32	int32	✗	native int	✗	& **`add.ovf.un`**	✗
int64	✗	int64	✗	✗	✗	✗
native int	native int	✗	native int	✗	& **`add.ovf.un`**	✗
F	✗	✗	✗	✗	✗	✗
&	& **`add.ovf.un`**, **`sub.ovf.un`**	✗	& **`add.ovf.un`**, **`sub.ovf.un`**	✗	native int **`sub.ovf.un`**	✗
O	✗	✗	✗	✗	✗	✗

These operations convert the top item on the evaluation stack 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 **`unsigned int16`**). 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 boxes marked ✗ indicate invalid CIL sequences.

Table 8: Conversion Operations

Convert-To	Input (from evaluation stack)					
	int32	**int64**	**native int**	**F**	**&**	**O**
int8 **unsigned int8** **int16** **unsigned int16**	Truncate[1]	Truncate[1]	Truncate[1]	Truncate to zero[2]	✗	✗
int32 **unsigned int32**	Nop	Truncate[1]	Truncate[1]	Truncate to zero[2]	✗	✗
int64	Sign extend	Nop	Sign extend	Truncate to zero[2]	Stop GC tracking	Stop GC tracking

unsigned int64	Zero extend	Nop	Zero extend	Truncate to zero[2]	Stop GC tracking	Stop GC tracking
native int	Sign extend	Truncate[1]	Nop	Truncate to zero[2]	Stop GC tracking	Stop GC tracking
native unsigned int	Zero extend	Truncate[1]	Nop	Truncate to zero[2]	Stop GC tracking	Stop GC tracking
All Float Types	To Float	To Float	To Float	Change precision[3]	✗	✗

1. "Truncate" means that the number is truncated to the desired size; ie, the most significant bytes of the input value are simply ignored. If the result is narrower than the minimum stack width of 4 bytes, then this result is zero extended (if the target type is unsigned) or sign-extended (if the target type is signed). Thus, converting the value 0x1234 ABCD from the evaluation stack to an 8-bit datum yields the result 0xCD; if the target type were int8, this is sign-extended to give 0xFFFF FFCD; if, instead, the target type were unsigned int8, this is zero-extended to give 0x0000 00CD.
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.
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 IEC 60559:1989 "round to nearest" mode to compute the low order bit of the result.
4. "Stop GC Tracking" means that, following the conversion, the item's value will *not* be reported to subsequent garbage-collection operations (and therefore will not be updated by such operations)

1.6 Implicit Argument Coercion

While the CLI operates only on 6 types (int32, native int, int64, F, O, and &) the metadata supplies a much richer model for parameters of methods. When about to call a method, the CLI performs implicit type conversions, detailed in the following table. (Conceptually, it inserts the appropriate `conv.*` instruction into the CIL stream, which may result in an information loss through truncation or rounding) This implicit conversion occurs for boxes marked ✓. Shaded boxes are not verifiable. Boxes marked ✗ indicate invalid CIL sequences. (A compiler is of course free to emit explicit `conv.*` **or** `conv.*.ovf` instructions to achieve any desired effect)

Table 9: Signature Matching

Type In Signature	**Stack Parameter**					
	int32	**native int**	**int64**	**F**	**&**	**O**
int8	✓	✓	✗	✗	✗	✗
unsigned int8, bool	✓	✓	✗	✗	✗	✗
int16	✓	✓	✗	✗	✗	✗
unsigned int16, char	✓	✓	✗	✗	✗	✗
int32	✓	✓	✗	✗	✗	✗
unsigned int32	✓	✓	✗	✗	✗	✗
int64	✗	✗	✓	✗	✗	✗
unsigned	✗	✗	✓	✗	✗	✗

int64						
native int	✓ Sign extend	✓	✘	✘	✘	✘
native unsigned int	✓ Zero extend	✓ Zero extend	✘	✘	✘	✘
float32	✘	✘	✘	Note[4]	✘	✘
float64	✘	✘	✘	Note[4]	✘	✘
Class	✘	✘	✘	✘	✘	✓
Value Type (Note[2])	Note[1]	Note[1]	Note[1]	Note[1]	✘	✘
By-Ref (&)	✘	✓ Start GC tracking	✘	✘	✓	✘
Ref Any (Note[3])	✘	✘	✘	✘	✘	✘

1. Passing a built-in type to a parameter that is required to be a value type is not allowed.
2. The CLI'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.
3. There are special instructions to construct and pass a **`Ref Any`**.
4. The CLI is permitted to pass floating point arguments using its internal F type, see clause 1.1.1. CIL generators may, of course, include an explicit conv.r4, conv.r4.ovf, or similar instruction.

Further notes concerning this table:

- On a 32-bit machine passing a **`native int`** argument to an **`unsigned int32`** parameter involves no conversion. On a 64-bit machine it is implicitly converted.
- "Start GC Tracking" means that, following the implicit conversion, the item's value will be reported to any subsequent garbage-collection operations, and perhaps changed as a result of the item pointed-to being relocated in the heap.

1.7 Restrictions on CIL Code Sequences

As well as detailed restrictions on CIL code sequences to ensure:

- Valid CIL
- Verifiable CIL

there are a few further restrictions, imposed to make it easier to construct a simple CIL-to-native-code compiler. This section specifies the general restrictions that apply in addition to this listed for individual instructions.

1.7.1 The Instruction Stream

The implementation of a method is provided by a contiguous block of CIL instructions, encoded as specified below. The address of the instruction block for a method as well as its length is specified in the file format (see Partition II, Common Intermediate Language Physical Layout). The first instruction is at the first byte (lowest address) of the instruction block.

Instructions are variable in size. The size of each instruction can be determined (decoded) from the content of the instruction bytes themselves. The size of and ordering of the bytes within an instruction is specified by each instruction definition. Instructions follow each other without padding in a stream of bytes that is both alignment and byte-order insensitive.

Each instruction occupies an exact number of bytes, and until the end of the instruction block, the next instruction begins immediately at the next byte. It is invalid for the instruction block (as specified by the block's length) to end without forming a complete last instruction.

Instruction prefixes extend the length of an instruction without introducing a new instruction; an instruction having one or more prefixes introduces only one instruction that begins at the first byte of the first instruction prefix.

Note: Until the end of the instruction block, the instruction following any control transfer instruction is decoded as an instruction and thus participates in locating subsequent instructions even if it is not the target of a branch. Only instructions may appear in the instruction stream, even if unreachable. There are no address-relative data addressing modes and raw data cannot be directly embedded within the instruction stream. Certain instructions allow embedding of immediate data as part of the instruction, however that differs from allowing raw data embedded directly in the instruction stream. Unreachable code may appear as the result of machine-generated code and is allowed, but it must always be in the form of properly formed instruction sequences.

The instruction stream can be translated and the associated instruction block discarded prior to execution of the translation. Thus, even instructions that capture and manipulate code addresses, such as `call`, `ret`, etc. can be virtualized to operate on translated addresses instead of addresses in the CIL instruction stream.

1.7.2 Valid Branch Targets

The set of addresses composed of the first byte of each instruction identified in the instruction stream defines the only valid instruction targets. Instruction targets include branch targets as specified in branch instructions, targets specified in exception tables such as protected ranges (see Partition I and Partition II), filter, and handler targets.

Branch instructions specify branch targets as either a 1-byte or 4-byte signed relative offset; the size of the offset is differentiated by the opcode of the instruction. The offset is defined as being relative to the byte following the branch instruction. [**Note**: Thus, an offset value of zero targets the immediately following instruction.]

The value of a 1-byte offset is computed by interpreting that byte as a signed 8-bit integer. The value of a 4-byte offset is can be computed by concatenating the bytes into a signed integer in the following manner: the byte of lowest address forms the least significant byte, and the byte with highest address forms the most significant byte of the integer. [**Note**: This representation is often called "a signed integer in little-endian byte-order".]

1.7.3 Exception Ranges

Exception tables describe ranges of instructions that are protected by catch, fault, or finally handlers (see Partition I and Partition II). The starting address of a protected block, filter clause, or handler shall be a valid branch target as specified in clause 1.7.2. It is invalid for a protected block, filter clause, or handler to end without forming a complete last instruction.

1.7.4 Must Provide Maxstack

Every method specifies a maximum number of items that can be pushed onto the CIL Evaluation. The value is stored in the `IMAGE_COR_ILMETHOD` structure that precedes the CIL body of each method. A method that specifies a maximum number of items less than the amount required by a static analysis of the method (using a traditional control flow graph without analysis of the data) is invalid (hence also unverifiable) and need not be supported by a conforming implementation of the CLI.

Note: Maxstack is related to analysis of the program, not to the size of the stack at runtime. It does not specify the maximum size in bytes of a stack frame, but rather the number of items that must be tracked by an analysis tool.

Rationale: *By analyzing the CIL stream for any method, it is easy to determine how many items will be pushed on the CIL Evaluation stack. However, specifying that maximum number ahead of time helps a CIL-to-native-code compiler (especially a simple one that does only a single pass through the CIL stream) in allocating internal data structures that model the stack and/or verification algorithm.*

1.7.5 Backward Branch Constraints

It must be possible, with a single forward-pass through the CIL instruction stream for any method, to infer the exact state of the evaluation stack at every instruction (where by "state" we mean the number and type of each item on the evaluation stack).

In particular, if that single-pass analysis arrives at an instruction, call it location X, that immediately follows an unconditional branch, and where X is not the target of an earlier branch instruction, then the state of the evaluation stack at X, clearly, cannot be derived from existing information. In this case, the CLI demands that the evaluation stack at X be empty.

Following on from this rule, it would clearly be invalid CIL if a later branch instruction to X were to have a non-empty evaluation stack

Rationale: *This constraint ensures that CIL code can be processed by a simple CIL-to-native-code compiler. It ensures that the state of the evaluation stack at the beginning of each CIL can be inferred from a single, forward-pass analysis of the instruction stream.*

Note: the stack state at location X in the above can be inferred by various means: from a previous forward branch to X; because X marks the start of an exception handler, etc.

See the following sections for further information:

- Exceptions: Partition I
- Verification conditions for branch instructions: Chapter 3
- The **`tail.`** prefix: Section 3.19

1.7.6 Branch Verification Constraints

The *target* of all branch instruction must be a valid branch target (see clause 1.7.2) within the method holding that branch instruction.

1.8 Verifiability

Memory safety is a property that ensures programs running in the same address space are correctly isolated from one another (see Partition I). 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 CLI 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 CIL code that is not verifiable, but which is known to be memory safe by the compiler writer. Thus, conforming CIL may not be verifiable, even though the producing compiler may *know* that it is memory safe. Several important uses of CIL 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 application programmer.

CIL 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 CIL. Verification tracks the types of values in much finer detail than is required for the basic functioning of the CLI, because it is checking that a CIL code sequence respects not only the basic rules of the CLI with respect to the safety of garbage collection, but also the typing rules of the CTS. This helps to guarantee the sound operation of the entire CLI.

The verifiability section of each operation description specifies requirements both for correct CIL generation and for verification. Correct CIL 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 CIL generation that are not captured in that diagram. Verification tests both the requirements for correct CIL generation and the specific verification conditions that are described with the instruction. The operation of CIL sequences that do not meet the CIL correctness requirements is unspecified. The operation of CIL sequences that meet the correctness requirements but are not verifiable may violate type safety and hence may violate security or memory access constraints.

1.8.1 Flow Control Restrictions for Verifiable CIL

This section specifies a verification algorithm that, combined with information on individual CIL instructions (see Chapter 3) and metadata validation (see Partition II), guarantees memory integrity.

The algorithm specified here creates a minimum level for all compliant implementations of the CLI in the sense that any program that is considered verifiable by this algorithm shall be considered verifiable and run correctly on all compliant implementations of the CLI.

The CLI provides a security permission (see Partition IV) that controls whether or not the CLI shall run programs that may violate memory safety. Any program that is verifiable according to this specification does not violate memory safety, and a conforming implementation of the CLI shall run such programs. The implementation may also run other programs provided it is able to show they do not violate memory safety (typically because they use a verification algorithm that makes use of specific knowledge about the implementation).

Note: While a compliant implementation is required to accept and run any program this verification algorithm states is verifiable, there may be programs that are accepted as verifiable by a given implementation but which this verification algorithm will fail to consider verifiable. Such programs will run in the given implementation but need not be considered verifiable by other implementations.

For example, an implementation of the CLI may choose to correctly track full signatures on method pointers and permit programs to execute the **calli** instruction even though this is not permitted by the verification algorithm specified here.

Implementers of the CLI are urged to provide a means for testing whether programs generated on their implementation meet this portable verifiability standard. They are also urged to specify where their verification algorithms are more permissive than this standard.

Only valid programs shall be verifiable. For ease of explanation, the verification algorithm described here assumes that the program is valid and does not explicitly call for tests of all validity conditions. Validity conditions are specified on a per-CIL instruction basis (see Chapter 3), and on the overall file format in Partition II.

1.8.1.1 Verification Algorithm

The verification algorithm shall attempt to associate a valid **`stack state`** with every CIL instruction. The stack state specifies the number of slots on the CIL stack at that point in the code and for each slot a required type that must be present in that slot. The initial stack state is empty (there are no items on the stack).

Verification assumes that the CLI zeroes all memory other than the evaluation stack before it is made visible to programs. A conforming implementation of the CLI shall provide this observable behavior. Furthermore, verifiable methods shall have the “zero initialize” bit set, see Partition II (Flags for Method Headers). If this bit is not set, then a CLI may throw a *Verification* exception at any point where a local variable is accessed, and where the assembly containing that method has not been granted *SecurityPermission.SkipVerification*

Rationale: *This requirement strongly enhances program portability, and a well-known technique (definite assignment analysis) allows a compiler from CIL to native code to minimize its performance impact. Note that a CLI may optionally choose to perform definite-assignment analysis – in such a case, it may confirm that a method, even without the “zero initialize” bit set, may in fact be verifiable (and therefore not throw a Verification exception)*

Note: Definite assignment analysis can be used by the CLI to determine which locations are written before they are read. Such locations needn’t be zeroed, since it isn’t possible to observe the contents of the memory as it was provided by the EE.

Performance measurements on C++ implementations (which does not require definite assignment analysis) indicate that adding this requirement has almost no impact, even in highly optimized code. Furthermore, customers incorrectly attribute bugs to the compiler when this zeroing is not performed, since such code often fails when small, unrelated changes are made to the program.

The verification algorithm shall simulate all possible control flow paths through the code and ensures that a legal stack state exists for every reachable CIL instruction. The verification algorithm does not take advantage of any data values during its simulation (e.g. it does not perform constant propagation), but uses only type assignments. Details of the type system used for verification and the algorithm used to merge stack states are provided in clause 1.8.1.3. The verification algorithm terminates as follows:

1. Successfully, when all control paths have been simulated.
2. Unsuccessfully when it is not possible to compute a valid stack state for a particular CIL instruction.
3. Unsuccessfully when additional tests specified in this clause fail.

There is a control flow path from every instruction to the subsequent instruction, with the exception of the unconditional branch instructions, **`throw`**, **`rethrow`**, and **`ret`**. Finally, there is a control flow path from each branch instruction (conditional or unconditional) to the branch target (targets, plural, for the **`switch`** instruction).

Verification simulates the operation of each CIL instruction to compute the new stack state, and any type mismatch between the specified conditions on the stack state (see Chapter 3) and the simulated stack state shall cause the verification algorithm to fail. (Note that verification simulates only the effect on the stack state: it does not perform the actual computation). The algorithm shall also fail if there is an existing stack state at the next instruction address (for conditional branches or instructions within a **try** block there may be more than one such address) that cannot be merged with the stack state just computed. For rules of this merge operation, see clause 1.8.1.3.

1.8.1.2 Verification Type System

The verification algorithm compresses types that are logically equivalent, since they cannot lead to memory safety violations. The types used by the verification algorithm are specified in clause 1.8.1.2.1, the type compatibility rules are specified in clause 1.8.1.2.2, and the rules for merging stack states are in clause 1.8.1.3.

1.8.1.2.1 Verification Types

The following table specifies the mapping of types used in the CLI and those used in verification. Notice that verification compresses the CLI types to a smaller set that maintains information about the size of those types in memory, but then compresses these again to represent the fact that the CLI stack expands 1, 2 and 4 byte built-in types into 4-byte types on the stack. Similarly, verification treats floating-point numbers on the stack as 64-bit quantities regardless of the actual representation.

Arrays are objects, but with special compatibility rules.

There is a special encoding for **`null`** that represents an object known to be the null value, hence with indeterminate actual type.

In the following table, “CLI Type” is the type as it is described in metadata. The “Verification Type” is a corresponding type used for type compatibility rules in verification (see clause 1.8.1.2.2) when considering the types of local variables, incoming arguments, and formal parameters on methods being called. The column “Verification Type (in stack state)” is used to simulate instructions that load data onto the stack, and shows the types that are actually maintained in the stack state information of the verification algorithm. The column “Managed Pointer to Type” shows the type tracked for managed pointers.

CLI Type	Verification Type	Verification Type (in stack state)	Managed Pointer to Type
`int8, unsigned int8, bool`	`int8`	`int32`	`& int8`
`int16, unsigned int16, char`	`int16`	`int32`	`& int16`
`int32, unsigned int32`	`int32`	`int32`	`& int32`
`int64, unsigned int64`	`int64`	`int64`	`& int64`
`native int, native unsigned int`	`native int`	`native int`	`& native int`
`float32`	`float32`	`float64`	`& float32`
`float64`	`float64`	`float64`	`& float64`

`Any value type`	`Same type`	`Same type`	`& Same type`
`Any object type`	`Same type`	`Same type`	`& Same type`
`Method pointer`	`Same type`	`Same type`	`Not valid`

A method can be defined as returning a managed pointer, but calls upon such methods are not verifiable.

Rationale: *some uses of returning a managed pointer are perfectly verifiable (eg, returning a reference to a field in an object); but some not (eg, returning a pointer to a local variable of the called method). Tracking this in the general case is a burden, and therefore not included in this standard.*

1.8.1.2.2 Verification Type Compatibility

The following rules define type compatibility. We use `S` and `T` to denote verification types, and the notation "`S := T`" to indicate that the verification type `T` can be used wherever the verification type `S` can be used, while "`S !:= T`" indicates that `T` cannot be used where `S` is expected. These are the verification type compatibility (see Partition I) rules. We use `T[]` to denote an array (of any rank) whose elements are of type `T`, and `T&` to denote a managed pointer to type `T`.

1. [`:=` is reflexive] For all verification types `S`, `S := S`
2. [`:=` is transitive] For all verification types `S`, `T`, and `U` if `S := T` and `T := U`, then `S := U`.
3. `S := T` if `S` is the base class of `T` or an interface implemented by `T` and `T` is not a value type.
4. `S := T` if `S` and `T` are both interfaces and the implementation of `T` requires the implementation of `S`
5. `S := null` if `S` is an object type or an interface
6. `S[] := T[]` if `S := T` and the arrays are either both vectors (zero-based, rank one) or neither is a vector and both have the same rank.
7. If `S` and `T` are method pointers, then `S := T` if the signatures (return types, parameter types, calling convention, and any custom attributes or custom modifiers) are the same.
8. Otherwise `S !:= T`

1.8.1.3 Merging Stack States

As the verification algorithm simulates all control flow paths it shall merge the simulated stack state with any existing stack state at the next CIL instruction in the flow. If there is no existing stack state, the simulated stack state is stored for future use. Otherwise the merge shall be computed as follows and stored to replace the existing stack state for the CIL instruction. If the merge fails, the verification algorithm shall fail.

The merge shall be computed by comparing the number of slots in each stack state. If they differ, the merge shall fail. If they match, then the overall merge shall be computed by merging the states slot-by-slot as follows. Let `T` be the type from the slot on the newly computed state and `S` be the type from the corresponding slot on the previously stored state. The merged type, `U`, shall be computed as follows (recall that `S := T` is the compatibility function defined in clause 1.8.1.2.2):

1. if `S := T` then `U=S`
2. Otherwise if `T := S` then `U=T`
3. Otherwise, if `S` and `T` are both object types, then let `V` be the closest common supertype of `S` and `T` then `U=V`.
4. Otherwise, the merge shall fail.

1.8.1.4 Class and Object Initialization Rules

The VES ensures that all statics are initially zeroed (i.e. built-in types are 0 or false, object references are null), hence the verification algorithm does not test for definite assignment to statics.

An object constructor shall not return unless a constructor for the base class or a different construct for the object's class has been called on the newly constructed object. The verification algorithm shall treat the `this` pointer as uninitialized unless the base class constructor has been called. No operations can be performed on an uninitialized **`this`** except for storing into and loading from the object's fields.

Note: If the constructor generates an exception the **`this`** pointer in the corresponding catch block is still uninitialized.

1.8.1.5 Delegate Constructors

The verification algorithm shall require that one of the following code sequences is used for constructing delegates; no other code sequence in verifiable code shall contain a **`newobj`** instruction for a delegate type. There shall be only one instance constructor method for a Delegate (overloading is not allowed)

The verification algorithm shall fail if a branch target is within these instruction sequences (other than at the start of the sequence).

Note: See Partition II for the signature of delegates and a validity requirement regarding the signature of the method used in the constructor and the signature of Invoke and other methods on the delegate class.

1.8.1.5.1 Delegating via Virtual Dispatch

The following CIL instruction sequence shall be used or the verification algorithm shall fail. The sequence begins with an object on the stack.

```
dup
ldvirtftn mthd   ; Method shall be on the class of the object,
          ; or one of its parent classes, or an interface
          ; implemented by the object
newobj delegateclass::.ctor(object, native int)
```

Rationale: *The* ***dup*** *is required to ensure that it is precisely the same object stored in the delegate as was used to compute the virtual method. If another object of a subtype were used the object and the method wouldn't match and could lead to memory violations.*

1.8.1.5.2 Delegating via Instance Dispatch

The following CIL instruction sequence shall be used or the verification algorithm shall fail. The sequence begins with either **`null`** or an object on the stack.

```
ldftn mthd              ; Method shall either be a static method or
          ; a method on the class of the object on the stack or
          ; one of the object's parent classes
newobj delegateclass::.ctor(object, native int)
```

1.9 Metadata Tokens

Many CIL instructions are followed by a "metadata token". This is a 4-byte value, that specifies a row in a metadata table, or a starting byte offset in the User String heap. The most-significant byte of the token specifies the table or heap. For example, a value of 0x02 specifies the TypeDef table; a value of 0x70 specifies the User String heap. The value corresponds to the number assigned to that metadata table (see Partition II for the full list of tables) or to 0x70 for the User String heap. The least-significant 3 bytes specify the target row within that metadata table, or starting byte offset within the User String heap. The rows within metadata tables are numbered one upwards, whilst offsets in the heap are numbered zero upwards. (So, for example, the metadata token with value 0x02000007 specifies row number 7 in the TypeDef table)

1.10 Exceptions Thrown

A CIL instruction can throw a range of exceptions. The CLI can also throw the general purpose exception called `ExecutionEngineException`. See Partition I for details.

2 Prefixes to Instructions

These special values are reserved to precede specific instructions. They do not constitute full instructions in their own right. It is not valid CIL to branch to the instruction following the prefix, but the prefix itself is a valid branch target. It is not valid CIL to have a prefix without immediately following it by one of the instructions it is permitted to precede.

2.1 tail. (prefix) – 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 is no longer required and thus can be removed before the call instruction is executed. Because the value returned by the call will be the value returned by this method, the call can be converted into a cross-method jump.

The evaluation 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 CIL must not branch to the `call` instruction, but it is permitted to branch to the `ret`. The only values on the stack must be the arguments for the method being called.

The `tail.call` (or `calli` or `callvirt`) instruction cannot be used to transfer control out of a try, filter, catch, or finally block. See Partition I.

The current frame cannot be discarded when control is transferred from untrusted code to trusted code, since this would jeopardize code identity security. 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.

CLI implementations are required to honor `tail. call` requests where caller and callee methods can be statically determined to lie in the same assembly; and where the caller is not in a synchronized region; and where caller and callee satisfy all conditions listed in the "Verifiability" rules below. (To "honor" the `tail.` prefix means to remove the caller's frame, rather than revert to a regular call sequence). Consequently, a CLI implementation need not honor `tail. calli` or `tail. callvirt` sequences.

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

Exceptions:

None.

Verifiability:

Correct CIL obeys the control transfer constraints listed above. In addition, no 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 managed pointers are passed to the method being called, since it does not track pointers into the current frame.

2.2 unaligned. (prefix) – pointer instruction may be unaligned

Format	Assembly Format	Description
FE 12 <**unsigned int8**>	unaligned. *alignment*	Subsequent pointer instruction may be unaligned

Stack Transition:

```
..., addr → ..., addr
```

Description:

`Unaligned.` specifies that *address* (an unmanaged pointer (**&**), or **`native int)`** on the stack may not be 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 Partition I and Partition II) must use **`unaligned.`** if the alignment is not known at compile time to be 8-byte.

The value of *alignment* shall be 1, 2, or 4 and means that the generated code should assume that *addr* is byte, double byte, or quad byte aligned, respectively.

Rationale: *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.

Note: See Partition I, 12.7 for information about atomicity and data alignment.

Exceptions:

None.

Verifiability:

An **`unaligned.`** prefix shall be immediately followed by one of the instructions listed above.

2.3 volatile. (prefix) - 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 referenced externally to the current thread of execution) 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. [see Partition I]

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:

A **`volatile.`** prefix should be immediately followed by one of the instructions listed above.

3 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 CTS’s object model are contained in the Object Model Instructions section.

3.1 add - add numeric values

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

Stack Transition:

`…, value1, value2` $\rightarrow$ `…, 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.

3.2 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 7: Overflow Arithmetic Operations.

Exceptions:

`OverflowException` is thrown if the result cannot be represented in the result type.

Verifiability:

See Table 7: Overflow Arithmetic Operations.

3.3 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 *value1* and *value2* 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.

3.4 arglist - get argument list

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

Stack Transition:

… $\rightarrow$ …, `argListHandle`

Description:

The **`arglist`** instruction returns an opaque handle (an unmanaged pointer, type **`native int`**) 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 CIL 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 verification requires that the result is an instance of the `System.RuntimeArgumentHandle` class.

3.5 beq.<length> – branch on equal

Format	Assembly Format	Description
3B <**int32**>	beq *target*	Branch to *target* if equal
2E <**int8**>	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.

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

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

Format	Assembly Format	Description
3C <**int32**>	bge *target*	Branch to *target* if greater than or equal to
2F <**int8**>	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.un`** 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 effect of a "**`bge`** *target"* instruction is identical to:

- If stack operands are integers, then : **`clt`** followed by a **`brfalse`** ***target***
- If stack operands are floating-point, then : **`clt.un`** followed by a **`brfalse`** ***target***

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

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

3.7 bge.un.<length> – branch on greater/equal, unsigned or unordered

Format	Assembly Format	Description
41 <**int32**>	bge.un *target*	Branch to *target* if greater than or equal to (unsigned or unordered)
34 <**int8**>	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`** 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.

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

3.8 bgt.<length> – branch on greater than

Format	Assembly Format	Description
3D <**int32**>	bgt *target*	Branch to *target* if greater than
30 <**int8**>	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.

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

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

Format	Assembly Format	Description
42 <**int32**>	bgt.un *target*	Branch to *target* if greater than (unsigned or unordered)
35 <**int8**>	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.

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

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

Format	Assembly Format	Description
3E <**int32**>	ble *target*	Branch to *target* if less than or equal to
31 <**int8**>	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*. *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 effect of a "**`ble`** *target*" instruction is identical to:

- If stack operands are integers, then : **`cgt`** followed by a **`brfalse`** ***target***
- If stack operands are floating-point, then : **`cgt.un`** followed by a **`brfalse`** ***target***

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

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

3.11 ble.un.<length> – branch on less/equal, unsigned or unordered

Format	**Assembly Format**	**Description**
43 <**int32**>	ble.un *target*	Branch to *target* if less than or equal to (unsigned or unordered)
36 <**int8**>	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). *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 effect of a "**`ble.un`** *target"* instruction is identical to:

- If stack operands are integers, then : **`cgt.un`** followed by a **`brfalse`** ***target***
- If stack operands are floating-point, then : **`cgt`** followed by a **`brfalse`** ***target***

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

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

3.12 blt.<length> – branch on less than

Format	**Assembly Format**	**Description**
3F <**int32**>	blt *target*	Branch to *target* if less than
32 <**int8**>	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.

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

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

Format	Assembly Format	Description
44 <**int32**>	blt.un *target*	Branch to *target* if less than (unsigned or unordered)
37 <**int8**>	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,* 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 **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.

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

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

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

Format	Assembly Format	Description
40 <**int32**>	bne.un *target*	Branch to *target* if unequal or unordered
33 <**int8**>	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.

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

3.15 br.<length> – unconditional branch

Format	Assembly Format	Description
38 <**int32**>	br *target*	Branch to *target*
2B <**int8**>	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.

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

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

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

3.16 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 CLI 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.

The **break** instruction may trap to a debugger, do nothing, or raise a security exception: the exact behavior is implementation-defined

Exceptions:

None.

Verifiability:

The **break** instruction is always verifiable.

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

Format	**Assembly Format**	**Description**
39 <**int32**>	brfalse *target*	Branch to *target* if *value* is zero (false)
2C <**int8**>	brfalse.s *target*	Branch to *target* if *value* is zero (false), short form
39 <**int32**>	brnull *target*	Branch to *target* if *value* is null (*alias for* **brfalse**)
2C <**int8**>	brnull.s *target*	Branch to *target* if *value* is null (*alias for* **brfalse.s**), short form
39 <**int32**>	brzero *target*	Branch to *target* if *value* is zero (*alias for* **brfalse**)
2C <**int8**>	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 type **int32, int64, object reference, managed pointer, unmanaged pointer** or **native int**) is zero (false). If *value* is non-zero (true) execution continues at the next instruction.

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.

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

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

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

Format	Assembly Format	Description
3A <**int32**>	brtrue *target*	Branch to *target* if *value* is non-zero (true)
2D <**int8**>	brtrue.s *target*	Branch to *target* if *value* is non-zero (true), short form
3A <**int32**>	brinst *target*	Branch to *target* if *value* is a non-null object reference (alias for **`brtrue`**)
2D <**int8**>	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 *target* if *value* (of type **`native int`**) 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.

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

Control transfers into and out of **`try`**, **`catch`**, **`filter`**, and **`finally`** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **`leave`** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 path to the destination instruction. See Section 1.5 for more details.

3.19 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 Partition II) 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 Partition I for a detailed description of the CIL 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 (see Section 2.1).

The metadata token carries sufficient information to determine whether the call is to a static method, an instance method, a virtual method, or a global function. In all of these cases the destination address is determined entirely from the metadata token (Contrast with the **callvirt** instruction for calling virtual methods, where the destination address also depends upon the runtime type of the instance reference pushed before the **callvirt**; see below).

If the method does not exist in the class specified by the metadata token, the base classes are searched to find the most derived class which defines the method and that method is called.

Rationale: *This implements "call superclass" behavior.*

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 signature carried in the metadata does not contain an entry in the parameter list for the **this** pointer but uses a bit (called HASTHIS) to indicate whether the method requires passing the **this** pointer (see Partition II)
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. Note that a delegate's **Invoke** method may be called with either the **call** or **callvirt** instruction.

Exceptions:

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

Verifiability:

Correct CIL 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, verification 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, and (d) the method is not abstract (ie, it has an implementation)

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 verification. A **call** annotated by **tail.** is also a special case.

If the target method is global (defined outside of any type), then the method must be static.

3.20 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 Partition I for a description of the CIL 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.

[A callee of "higher trust" is defined as one whose permission grant-set is a strict superset of the grant-set of the caller.]

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). 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 Partition II) 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 system security does not grant the caller access to the called method. The security check may occur when the CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL 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.

Verification checks that *ftn* is a pointer to a function generated by **`ldftn`** or **`ldvirtfn`**.

3.21 ceq - compare equal

Format	Assembly Format	Description
FE 01	ceq	Push 1 (of type **int32**) 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 **int32**) is pushed on the stack. Otherwise 0 (of type **int32**) is pushed on the stack.

For floating-point numbers, **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 CIL 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.

3.22 cgt - compare greater than

Format	Assembly Format	Description
FE 02	cgt	Push 1 (of type **`int32`**) 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 **`int32`**) is pushed on the stack. Otherwise 0 (of type **`int32`**) 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 IEC 60559:1989, 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 CIL 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.

3.23 cgt.un - compare greater than, unsigned or unordered

Format	Assembly Format	Description
FE 03	cgt.un	Push 1 (of type **`int32`**) 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 **`int32`**) is pushed on the stack if

- for floating-point numbers, either *value1* is strictly greater than *value2*, or *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 **`int32`**) is pushed on the stack.

As per IEC 60559:1989, 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 CIL 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.

3.24 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 $\rightarrow$ …, value

Description:

The **`ckfinite`** 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.

Verifiability:

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

3.25 clt - compare less than

Format	Assembly Format	Description
FE 04	clt	Push 1 (of type **`int32`**) 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 **`int32`**) is pushed on the stack. Otherwise 0 (of type **`int32`**) 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 IEC 60559:1989, 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 CIL 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.

3.26 clt.un - compare less than, unsigned or unordered

Format	Assembly Format	Description
FE 05	clt.un	Push 1 (of type **`int32`**) 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 **`int32`**) is pushed on the stack if

- for floating-point numbers, either *value1* is strictly less than *value2*, or *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 **`int32`**) is pushed on the stack.

As per IEC 60559:1989, 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 CIL 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.

3.27 conv.<to type> - data conversion

Format	Assembly Format	Description
67	conv.i1	Convert to **`int8`**, pushing **`int32`** on stack
68	conv.i2	Convert to **`int16`**, pushing **`int32`** on stack
69	conv.i4	Convert to **`int32`**, pushing **`int32`** on stack
6A	conv.i8	Convert to **`int64`**, pushing **`int64`** on stack
6B	conv.r4	Convert to **`float32`**, pushing **`F`** on stack
6C	conv.r8	Convert to **`float64`**, pushing **`F`** on stack
D2	conv.u1	Convert to **`unsigned int8`**, pushing **`int32`** on stack
D1	conv.u2	Convert to **`unsigned int16`**, pushing **`int32`** on stack
6D	conv.u4	Convert to **`unsigned int32`**, pushing **`int32`** on stack
6E	conv.u8	Convert to **`unsigned int64`**, pushing **`int64`** on stack
D3	conv.i	Convert to **`native int`**, pushing **`native int`** on stack
E0	conv.u	Convert to **`native unsigned int`**, pushing **`native int`** 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 values of less than 4 bytes are extended to **`int32`** (not **`native int`**) 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 a **`float64`** to a **`float32`**, precision may be lost. If *value* is too large to fit in a **`float32`**, the IEC 60559:1989 positive infinity (if *value* is positive) or IEC 60559:1989 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 **`int32`**, 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 integer off the stack, interprets it as unsigned, and replaces it with a floating-point number to represent the integer; either a **`float32`**, if this is wide enough to represent the integer without loss of precision, else a **`float64`**.

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

The acceptable operand types and their corresponding result data type is encapsulated in Table 8: Conversion Operations.

Exceptions:

None.

Verifiability:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

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

Format	Assembly Format	Description
B3	conv.ovf.i1	Convert to an **int8** (on the stack as **int32**) and throw an exception on overflow
B5	conv.ovf.i2	Convert to an **int16** (on the stack as **int32**) and throw an exception on overflow
B7	conv.ovf.i4	Convert to an **int32** (on the stack as **int32**) and throw an exception on overflow
B9	conv.ovf.i8	Convert to an **int64** (on the stack as **int64**) and throw an exception on overflow
B4	conv.ovf.u1	Convert to a **unsigned int8** (on the stack as **int32**) and throw an exception on overflow
B6	conv.ovf.u2	Convert to a **unsigned int16** (on the stack as **int32**) and throw an exception on overflow
B8	conv.ovf.u4	Convert to a **unsigned int32** (on the stack as **int32**) and throw an exception on overflow
BA	conv.ovf.u8	Convert to a **unsigned int64** (on the stack as **int64**) and throw an exception on overflow
D4	conv.ovf.i	Convert to an **native int** (on the stack as **native int**) and throw an exception on overflow
D5	conv.ovf.u	Convert to a **native unsigned int** (on the stack as **native int**) 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 is too large or too small to be represented by the target type, an exception is thrown.

Conversions from floating-point numbers to integral values truncate the number toward zero. Note that integer values of less than 4 bytes are extended to **int32** (not **native int**) on the evaluation stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 8: Conversion Operations.

Exceptions:

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

Verifiability:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

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

Format	Assembly Format	Description
82	conv.ovf.i1.un	Convert unsigned to an **`int8`** (on the stack as **`int32`**) and throw an exception on overflow
83	conv.ovf.i2.un	Convert unsigned to an **`int16`** (on the stack as **`int32`**) and throw an exception on overflow
84	conv.ovf.i4.un	Convert unsigned to an **`int32`** (on the stack as **`int32`**) and throw an exception on overflow
85	conv.ovf.i8.un	Convert unsigned to an **`int64`** (on the stack as **`int64`**) and throw an exception on overflow
86	conv.ovf.u1.un	Convert unsigned to an **`unsigned int8`** (on the stack as **`int32`**) and throw an exception on overflow
87	conv.ovf.u2.un	Convert unsigned to an **`unsigned int16`** (on the stack as **`int32`**) and throw an exception on overflow
88	conv.ovf.u4.un	Convert unsigned to an **`unsigned int32`** (on the stack as **`int32`**) and throw an exception on overflow
89	conv.ovf.u8.un	Convert unsigned to an **`unsigned int64`** (on the stack as **`int64`**) and throw an exception on overflow
8A	conv.ovf.i.un	Convert unsigned to a **`native int`** (on the stack as **`native int`**) and throw an exception on overflow
8B	conv.ovf.u.un	Convert unsigned to a **`native unsigned int`** (on the stack as **`native int`**) 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 values of less than 4 bytes are extended to **`int32`** (not **`native int`**) on the evaluation stack.

The acceptable operand types and their corresponding result data type is encapsulated in Table 8: Conversion Operations.

Exceptions:

`OverflowException` is thrown if the result cannot be represented in the result type

Verifiability:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack. The same table specifies a restricted set of types that are acceptable in verified code.

3.30 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 `unsigned int32`) bytes from address *srcaddr* (of type `native int`, or `&`) to address *destaddr* (of type `native int`, 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.

Rationale: *`cpblk` is intended for copying structures (rather than arbitrary byte-runs). All such structures, allocated by the CLI, are naturally aligned for the current platform. Therefore, there is no need for the compiler that generates `cpblk` instructions to be aware of whether the code will eventually execute on a 32-bit or 64-bit platform.*

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

Exceptions:

`NullReferenceException` may be thrown if an invalid address is detected.

Verifiability:

The `cpblk` instruction is never verifiable. Correct CIL ensures the conditions specified above.

3.31 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:

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

|*result*| = |*value1*| / |*value2*|, and

sign(result) = +, if sign(value1) = sign(value2), or
–, if sign(value1) ~= sign(value2)

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

Integer division truncates towards zero.

Floating-point division is per IEC 60559:1989. 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 cannot be represented in the result type. This can happen if *value1* is the smallest representable integer value, and *value2* is -1.

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

Floating-point operations never throw an exception (they produce NaNs or infinities instead, see Partition I).

Example:

`+14 div +3 is 4`

`+14 div -3 is -4`

`-14 div +3 is -4`

`-14 div -3 is 4`

Verifiability:

See Table 2: Binary Numeric Operations.

3.32 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.un** 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 are encapsulated in Table 5: Integer Operations.

Exceptions:

`DivideByZeroException` is thrown if *value2* is zero.

Example:

```
+5 div.un +3      is 1
+5 div.un -3      is 0
-5 div.un +3      is 14316557630 or 0x55555553
-5 div.un -3      is 0
```

Verifiability:

See Table 5: Integer Operations.

3.33 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.

3.34 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 Partition I for a discussion of exceptions). *Value* (which must be of type **int32** and is one of a specific set of values) is returned from the **filter** clause. It should be one of:

- **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 CLI 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 **filter** block. It is not legal to embed a **try** block within a **filter** block. If an exception is thrown inside the **filter** block, it is intercepted and a value of **exception_continue_search** is returned.

Exceptions:

None.

Verifiability:

Correct CIL guarantees the control transfer restrictions specified above. Also, the stack must contain exactly one item (of type **int32**).

3.35 endfinally – end the finally or fault clause of an exception block

Format	Assembly Format	Description
DC	Endfault	End fault clause of an exception block
DC	Endfinally	End finally clause of an exception block

Stack Transition:

… $\rightarrow$ …

Description:

Return from the `finally` or `fault` clause of an exception block; see the Exception Handling section of Partition I for details.

Signals the end of the `finally` or `fault` clause so that stack unwinding can continue until the exception handler is invoked. The `endfinally` or `endfault` instruction transfers control back to the CLI exception mechanism. This then searches for the next `finally` clause in the chain, if the protected block was exited with a `leave` instruction. If the protected block was exited with an exception, the CLI will search for the next `finally` or `fault`, or enter 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. These same restrictions apply to the `endfault` instruction and the `fault` block, *mutatis mutandis*.

Control cannot be transferred into a `finally` (or `fault` block) except through the exception mechanism. Control cannot be transferred out of a `finally` (or `fault)` block except through the use of a `throw` instruction or executing the `endfinally` (or `endfault`) instruction. In particular, it is not legal to "fall out" of a `finally` (or `fault`) block or to execute a `ret` or `leave` instruction within a `finally` (or `fault`) block.

Note that the `endfault` and `endfinally` instructions are aliases – they correspond to the same opcode.

Exceptions:

None.

Verifiability:

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

3.36 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 **unsigned int32**) bytes starting at *addr* (of type **native int**, or **&**) to *value* (of type **unsigned int8**). **initblk** assumes that *addr* is aligned to the natural size of the machine (but see the **unaligned.** prefix instruction).

Rationale: ***initblk*** *is intended for initializing structures (rather than arbitrary byte-runs). All such structures, allocated by the CLI, are naturally aligned for the current platform. Therefore, there is no need for the compiler that generates* ***initblk*** *instructions to be aware of whether the code will eventually execute on a 32-bit or 64-bit platform.*

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

Exceptions:

NullReferenceException may be thrown if an invalid address is detected.

Verifiability:

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

3.37 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 Partition II). 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, fault or finally block; or out of a synchronized region. If this is done, results are undefined. See Partition I.

Exceptions:

None.

Verifiability:

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

3.38 ldarg.<length> - load argument onto the stack

Format	Assembly Format	Description
FE 09 <*unsigned int16*>	ldarg *num*	Load argument numbered *num* onto stack.
0E <*unsigned int8*>	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 *num*'th incoming argument, where arguments are numbered 0 onwards (see Partition I) onto the evaluation stack. The **ldarg** instruction can be used to load a value type or a built-in 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 of the first 4 arguments. The **ldarg.s** instruction is an efficient encoding for loading argument numbers 4 through 255.

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. (See the **arglist** instruction)

Arguments that hold an integer value smaller than 4 bytes long are expanded to type **int32** when they are loaded onto the stack. Floating-point values are expanded to their native size (type **F**).

Exceptions:

None.

Verifiability:

Correct CIL guarantees that *num* is a valid argument index. See Section 1.5 for more details on how verification determines the type of the value loaded onto the stack.

3.39 ldarga.<length> - load an argument address

Format	Assembly Format	Description
FE 0A <*unsigned int16*>	ldarga *argNum*	fetch the address of argument *argNum*.
0F <*unsigned int8*>	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. managed pointer) of the *argNum*'th argument, where arguments are numbered 0 onwards. 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 argument numbers 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 Partition I). In other cases, `ldarg` and `starg` should be used.*

Exceptions:

None.

Verifiability:

Correct CIL ensures that *argNum* is a valid argument index. See Section 1.5 for more details on how verification determines the type of the value loaded onto the stack.

3.40 ldc.<type> - load numeric constant

Format	Assembly Format	Description
20 <`int32`>	ldc.i4 *num*	Push *num* of type `int32` onto the stack as `int32`.
21 <`int64`>	ldc.i8 *num*	Push *num* of type `int64` onto the stack as `int64`.
22 <`float32`>	ldc.r4 *num*	Push *num* of type `float32` onto the stack as `F`.
23 <`float64`>	ldc.r8 *num*	Push *num* of type `float64` onto the stack as `F`.
16	ldc.i4.0	Push 0 onto the stack as `int32`.
17	ldc.i4.1	Push 1 onto the stack as `int32`.
18	ldc.i4.2	Push 2 onto the stack as `int32`.
19	ldc.i4.3	Push 3 onto the stack as `int32`.
1A	ldc.i4.4	Push 4 onto the stack as `int32`.
1B	ldc.i4.5	Push 5 onto the stack as `int32`.
1C	ldc.i4.6	Push 6 onto the stack as `int32`.
1D	ldc.i4.7	Push 7 onto the stack as `int32`.
1E	ldc.i4.8	Push 8 onto the stack as `int32`.
15	ldc.i4.m1	Push -1 onto the stack as `int32`.
15	ldc.i4.M1	Push -1 of type `int32` onto the stack as `int32` (alias for `ldc.i4.m1`).
1F <`int8`>	ldc.i4.s *num*	Push *num* onto the stack as `int32`, 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, as well as 4-byte values that do not fit in the short forms.

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 IEC 60559:1989 number, since these representations are not portable across architectures.

Exceptions:

None.

Verifiability:

The `ldc` instruction is always verifiable.

3.41 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 **`native int`**) to the native code implementing the method described by *method* (a metadata token, either a **`methoddef`** or **`methodref`**; see Partition II) 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 CIL interpreter when a native version of the method isn't available).

Exceptions:

None.

Verifiability:

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

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

Format	Assembly Format	Description
46	ldind.i1	Indirect load value of type **`int8`** as **`int32`** on the stack.
48	ldind.i2	Indirect load value of type **`int16`** as **`int32`** on the stack.
4A	ldind.i4	Indirect load value of type **`int32`** as **`int32`** on the stack.
4C	ldind.i8	Indirect load value of type **`int64`** as **`int64`** on the stack.
47	ldind.u1	Indirect load value of type **`unsigned int8`** as **`int32`** on the stack.
49	ldind.u2	Indirect load value of type **`unsigned int16`** as **`int32`** on the stack.
4B	ldind.u4	Indirect load value of type **`unsigned int32`** as **`int32`** on the stack.
4E	ldind.r4	Indirect load value of type **`float32`** as **`F`** on the stack.
4C	ldind.u8	Indirect load value of type **`unsigned int64`** as **`int64`** on the stack (alias for **`ldind.i8`**).
4F	ldind.r8	Indirect load value of type **`float64`** as **`F`** on the stack.
4D	ldind.i	Indirect load value of type **`native int`** as **`native int`** on the stack
50	ldind.ref	Indirect load value of type **`object ref`** as **`O`** on the stack.

Stack Transition:

…, addr → …, value

Description:

The **`ldind`** instruction indirectly loads a value from address *addr* (an unmanaged pointer, **`native int`**, or managed 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 values of less than 4 bytes are extended to **`int32`** (not **`native int`**) when they are loaded onto the evaluation stack. Floating-point values are converted to **`F`** type when loaded onto the evaluation stack.

Correct CIL ensures that the **`ldind`** instructions are 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 a `NullReferenceException` may occur (but see the **`unaligned.`** prefix instruction). The results of all CIL 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 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 CLI can know whether to sign extend or zero extend. The **`ldind.u8`** and **`ldind.u4`** variants are provided for convenience; **`ldind.u8`** is an alias for **`ldind.i8`**; **`ldind.u4`** and **`ldind.i4`** have different opcodes, but their effect is identical*

Exceptions:

`NullReferenceException` may be thrown if an invalid address is detected.

Verifiability:

Correct CIL only uses an **`ldind`** instruction in a manner consistent with the type of the pointer.

3.43 ldloc - load local variable onto the stack

Format	Assembly Format	Description
FE 0C<*unsigned int16*>	ldloc *indx*	Load local variable of index *indx* onto stack.
11 <*unsigned int8*>	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:

… $\rightarrow$ …, value

Description:

The **ldloc** *indx* instruction pushes the contents of the local variable number *indx* onto the evaluation stack, where local variables are numbered 0 onwards. Local variables are initialized to 0 before entering the method only if the initialize flag on the method is true (see Partition I). 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 local variables 4 through 255.

The type of the value is the same as the type of the local variable, which is specified in the method header. See Partition I.

Local variables that are smaller than 4 bytes long are expanded to type **int32** when they are loaded onto the stack. Floating-point values are expanded to their native size (type **F**).

Exceptions:

VerificationException is thrown if the the "zero initialize" bit for this method has not been set, and the assembly containing this method has not been granted SecurityPermission.SkipVerification (and the CIL does not perform automatic definite-assignment analysis)

Verifiability:

Correct CIL ensures that *indx* is a valid local index. See Section 1.5 for more details on how verification determines the type of a local variable. For the *ldloca indx* instruction, i*ndx* must lie in the range 0 to 65534 inclusive (specifically, 65535 is not valid)

Rationale: *The reason for excluding 65535 is pragmatic: likely implementations will use a 2-byte integer to track both a local's index, as well as the total number of locals for a given method. If an index of 65535 had been made legal, it would require a wider integer to track the number of locals in such a method.*

Also, for verifiable code, this instruction must guarantee that it is not loading an uninitialized value – whether that initialization is done explicitly by having set the "zero initialize" bit for the method, or by previous instructions (where the CLI performs definite-assignment analysis)

3.44 ldloca.<length> - load local variable address

Format	**Assembly Format**	**Description**
FE 0D <*unsigned int16*>	ldloca *index*	Load address of local variable with index *indx*
12 <*unsigned int8*>	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 number *index* onto the stack, where local variables are numbered 0 onwards. The value pushed on the stack is already aligned correctly for use with instructions like **`ldind`** and **`stind`**. The result is a managed pointer (type **`&`**). The **`ldloca.s`** instruction provides an efficient encoding for use with the local variables 0 through 255.

Exceptions:

`VerificationException` is thrown if the the "zero initialize" bit for this method has not been set, and the assembly containing this method has not been granted SecurityPermission.SkipVerification (and the CIL does not perform automatic definite-assignment analysis)

Verifiability:

Correct CIL ensures that *indx* is a valid local index. See Section 1.5 for more details on how verification determines the type of a local variable. For the *ldloca indx* instruction, i*ndx* must lie in the range 0 to 65534 inclusive (specifically, 65535 is not valid)

Rationale: *The reason for excluding 65535 is pragmatic: likely implementations will use a 2-byte integer to track both a local's index, as well as the total number of locals for a given method. If an index of 65535 had been made legal, it would require a wider integer to track the number of locals in such a method.*

Also, for verifiable code, this instruction must guarantee that it is not loading an uninitialized value – whether that initialization is done explicitly by having set the "zero initialize" bit for the method, or by previous instructions (where the CLI performs definite-assignment analysis)

3.45 ldnull – load a null pointer

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

Stack Transition:

`… → …, null value`

Description:

The `ldnull` pushes a null reference (type `O`) on the stack. This is used to initialize locations before they become live or when they become dead.

Rationale: *It might be thought that* `ldnull` *is redundant: why not use* `ldc.i4.0` *or* `ldc.i8.0` *instead? The answer is that* `ldnull` *provides a size-agnostic null – analogous to a* `ldc.i` *instruction, which does not exist. However, even if CIL were to include a* `ldc.i` *instruction it would still benefit verification algorithms to retain the* `ldnull` *instruction because it makes type tracking easier.*

Exceptions:

None.

Verifiability:

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

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

Format	Assembly Format	Description
DD <**int32**>	leave *target*	Exit a protected region of code.
DE <**int8**>	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 in such a block to transfer control within it. The **leave** instruction empties the evaluation stack and 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 CIL requires the computed destination lie within the current method. See Section 1.5 for more details.

3.47 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` $\rightarrow$ `address`

Description:

The **`localloc`** instruction allocates *size* (type **`native unsigned int`**) bytes from the local dynamic memory pool and returns the address (a managed 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 Partition I). 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 built-in data type can be stored there using the **`stind`** instructions and loaded using the **`ldind`** instructions.

The **`localloc`** instruction cannot occur within an exception block: **`filter`**, **`catch`**, **`finally`**, or **`fault.`**

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:

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

Verifiability:

Correct CIL requires that the evaluation stack be empty, apart from the *size* item. This instruction is never verifiable.

3.48 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 types are encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 2: Binary Numeric Operations.

3.49 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 types are encapsulated in Table 7: Overflow Arithmetic Operations.

Exceptions:

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

Verifiability:

See Table 8: Conversion Operations.

3.50 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 types are encapsulated in Table 3: Unary Numeric Operations.

Exceptions:

None.

Verifiability:

See Table 3: Unary Numeric Operations.

3.51 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.

3.52 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.

3.53 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.

3.54 pop – remove the top element of the stack

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

Stack Transition:

…, value $\rightarrow$ …

Description:

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

Exceptions:

None.

Verifiability:

No additional requirements.

3.55 rem - compute remainder

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

Stack Transition:

…, value1, value2 $\rightarrow$ …, result

Description:

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

For integer operands

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

result = *value1* – *value2*×(*value1* **div** *value2*), and

0 ≤ |*result*| < |*value2*|, and

sign(*result*) = *sign*(*value1*),

where `div` is the division instruction, which truncates towards zero.

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

For floating-point operands

`rem` is defined similarly, except that, if *value2* is zero or *value1* is infinity the result is NaN. If *value2* is `infinity`, the result is *value1* (negated for –`infinity`). This definition is different from the one for floating-point remainder in the IEC 60559:1989 Standard. That Standard specifies that *value1* ***div*** *value2* is the nearest integer instead of truncating towards zero. `System.Math.IEEERemainder` (see Partition IV) provides the IEC 60559:1989 behavior.

Exceptions:

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

Integral operations may throw `ArithmeticException` if *value1* is the smallest representable integer value and *value2* is -1.

Example:

```
+10 rem +6 is 4    (+10 div +6 = 1)
+10 rem -6 is 4    (+10 div -6 = -1)
-10 rem +6 is -4   (-10 div +6 = -1)
-10 rem -6 is -4   (-10 div -6 = 1)
```

For the various floating-point values of 10.0 and 6.0, ***rem*** gives the same values; `System.Math.IEEERemainder`, however, gives the following values.

```
System.Math.IEEERemainder(+10.0,+6.0) is -2    (+10.0 div +6.0 =  1.666…7)
System.Math.IEEERemainder(+10.0,-6.0) is -2    (+10.0 div -6.0 = -1.666…7)
System.Math.IEEERemainder(-10.0,+6.0) is  2    (-10.0 div +6.0 = -1.666…7)
System.Math.IEEERemainder(-10.0,-6.0) is  2    (-10.0 div -6.0 =  1.666…7)
```

Verifiability:

See Table 2: Binary Numeric Operations.

3.56 rem.un - compute integer remainder, unsigned

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

Stack Transition:

…, value1, value2 $\rightarrow$ …, result

Description:

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

result = *value1* – *value2*×(*value1* **div.un** *value2*), and

$0 \leq result < value2$,

where `div.un` is the unsigned division instruction. 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 are encapsulated in Table 5: Integer Operations.

Exceptions:

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

Example:

```
+5 rem.un +3     is 2                  (+5 div.un +3 = 1)
+5 rem.un -3     is 5                  (+5 div.un -3 = 0)
-5 rem.un +3     is 2                  ( -5 div.un +3 = 1431655763 or 0x55555553)
-5 rem.un -3     is -5 or 0xfffffffb   ( -5 div.un -3 = 0)
```

Verifiability:

See Table 5: Integer Operations.

3.57 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, if any, 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 generated CIL does not perform a method return from within a **`filter`** or **`finally`**. See Partition I.

Exceptions:

None.

Verifiability:

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

3.58 shl - shift integer left

Format	Assembly Format	Description
62	shl	Shift an integer left (shifting in zeros), return an integer

Stack Transition:

`…, value, shiftAmount` → `…, result`

Description:

The **`shl`** instruction shifts *value* (**`int32`**, **`int64`** or **`native int`**) left by the number of bits specified by *shiftAmount*. *shiftAmount* is of type **`int32`**, **`int64`** or **`native int`**. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. See Table 6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3.59 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* (**int32**, **int64** or **native int**) right by the number of bits specified by *shiftAmount*. *shiftAmount* is of type **int32**, **int64** or **native int**. 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. See Table 6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

3.60 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* (**`int32`**, **`int 64`** or **`native int`**) right by the number of bits specified by *shiftAmount*. *shiftAmount* is of type **`int32`** or **`native int`**. 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. See Table 6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Verifiability:

See Table 5: Integer Operations.

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

Format	**Assembly Format**	**Description**
FE 0B <**unsigned int16**>	starg *num*	Store a value to the argument numbered *num*
10 <**unsigned int8**>	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 Partition I). 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 CIL requires that *num* is a valid argument slot.

Verification also checks that the verification type of *value* matches the type of the argument, as specified in the current method's signature (verification types are less detailed than CLI types).

3.62 stind.<type> - store value indirect from stack

Format	Assembly Format	Description
52	stind.i1	Store value of type **`int8`** into memory at address
53	stind.i2	Store value of type **`int16`** into memory at address
54	stind.i4	Store value of type **`int32`** into memory at address
55	stind.i8	Store value of type **`int64`** into memory at address
56	stind.r4	Store value of type **`float32`** into memory at address
57	stind.r8	Store value of type **`float64`** into memory at address
DF	stind.i	Store value of type **`native int`** 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 **`native int`**, or managed pointer, type **`&`**). The address specified by *addr* must be aligned to the natural size of *val* or a `NullReferenceException` may occur (but see the **`unaligned.`** prefix instruction). The results of all CIL 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 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.

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

Exceptions:

`NullReferenceException` is thrown if *addr* is not naturally aligned for the argument type implied by the instruction suffix

Verifiability:

Correct CIL ensures that *addr* be a pointer whose type is known and is assignment compatible with that of *val*.

3.63 stloc - pop value from stack to local variable

Format	Assembly Format	Description
FE 0E <*unsigned int16*>	stloc *indx*	Pop value from stack into local variable *indx*.
13 <*unsigned int8*>	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 number *indx* (see Partition I), where local variables are numbered 0 onwards. 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 local variables 4 through 255.

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

Exceptions:

None.

Verifiability:

Correct CIL requires that *indx* is a valid local index. For the *stloc indx* instruction, i*ndx* must lie in the range 0 to 65534 inclusive (specifically, 65535 is not valid)

Rationale: *The reason for excluding 65535 is pragmatic: likely implementations will use a 2-byte integer to track both a local's index, as well as the total number of locals for a given method. If an index of 65535 had been made legal, it would require a wider integer to track the number of locals in such a method.*

Verification also checks that the verification type of *value* matches the type of the local, as specified in the current method's locals signature.

3.64 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 Table2: Binary Numeric Operations.

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

Format	Assembly Format	Description
DA	sub.ovf	Subtract `native int` from a `native int`. Signed result must fit in same size
DB	sub.ovf.un	Subtract native `unsigned int` from a `native unsigned int`. 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 7: Overflow Arithmetic Operations.

Exceptions:

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

Verifiability:

See Table 7: Overflow Arithmetic Operations.

3.66 switch – table switch on value

Format	Assembly Format	Description
45 *<unsigned int32> <int32>… <int32>*	switch *(t1, t2 … tn)*	jump to one of n values

Stack Transition:

…, value → …,

Description:

The **switch** instruction implements a jump table. The format of the instruction is an **unsigned int32** representing the number of targets *N*, followed by *N* **int32** values specifying jump targets: these targets are represented as offsets (positive or negative) from the beginning of the instruction following this switch instruction.

The switch instruction pops *value* off the stack and compares it, as an unsigned integer, to *N*. If *value* is less than *N*, execution is transferred to the *value*'th target, where targets are numbered from 0 (i.e., a *value* of 0 takes the first target, a *value* of 1 takes the second target, etc). If *value* is not less than *N*, execution continues at the next instruction (fall through).

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

Control transfers into and out of **try**, **catch**, **filter**, and **finally** blocks cannot be performed by this instruction. (Such transfers are severely restricted and must use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Verifiability:

Correct CIL 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 Section 1.5 for more details.

3.67 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 *value1* and *value2* 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.

4 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 built-in 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 common type system. 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 Partition I.

4.1 box – convert value type to object reference

Format	Assembly Format	Description
8C <*T*>	box *valTypeTok*	Convert *valueType* to a true object reference

Stack Transition:

`…, valueType` → `…, obj`

Description:

A value type has two separate representations (see Partition I) within the CLI:

- A ‘raw’ form used when a value type is embedded within another object or on the stack.
- 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 **`box`** instruction converts the ‘raw’ *valueType* (an unboxed value type) into an instance of type Object (of type **`O`**). This is accomplished by creating a new object and copying the data from *valueType* into the newly allocated object. *ValTypeTok* is a metadata token (a **`typeref`** or **`typedef`**) indicating the type of *valueType* (See Partition II)

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 CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that *valueType* is of the correct value type, and that *valTypeTok* is a **`typeref`** or **`typedef`** metadata token for that value type.

4.2 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 and instance methods. See Partition I for a detailed description of the CIL 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 original method the stack frame will not be released.

[A callee of "higher trust" is defined as one whose permission grant-set is a strict superset of the grant-set of the caller]

method is a metadata token (a **`methoddef`** or **`methodref`**; see Partition II) 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. The signature carried in the metadata does not contain an entry in the parameter list for the **`this`** pointer, but uses a bit (called HASTHIS) to indiciate whether the method requires passing the this pointer (see Partition II)

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

Exceptions:

`MissingMethodException` is 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 CIL is converted to native code, rather than at runtime.

`NullReferenceException` is thrown if *obj* is null.

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

Verifiability:

Correct CIL 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 arguments on the stack are consistent with the types expected by the method call, and (d) the method is accessible from the callsite. A **`callvirt`** annotated by **`tail.`** has additional considerations – see Section 1.5.

4.3 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`**), 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.

Note that:

1. Arrays inherit from `System.Array`
2. If Foo can be cast to Bar, then Foo[] can be cast to Bar[]
3. For the purposes of 2., enums are treated as their undertlying type: thus E1[] can cast to E2[] if E1 and E2 share an underlying type

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

Exceptions:

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

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

Verifiability:

Correct CIL ensures that *class* is a valid **`typeRef`** or **`typeDef`** token, and that *obj* is always either null or an object reference.

4.4 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, **`native int`**, 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`**), or if *classTok* does not represent a value type.

Exceptions:

`NullReferenceException` may be thrown if an invalid address is detected.

Verifiability:

Correct CIL 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 managed pointers (not unmanaged pointers).

4.5 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 **`native int`**, or **`&`**) to **`null`** or a 0 of the appropriate built-in type. After this method is called, the instance is ready for the constructor method to be called. Behavior is unspecified if either *addrOfValObj* is not a pointer to an instance of the class represented by *classTok* (a **`typeref`** or **`typedef;`** see Partition II), 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 CIL ensures that *classTok* is a valid **`typeref`** or **`typedef`** token specifying a value type, and that *valObj* is a managed pointer to an instance of that value type.

4.6 isinst – test if an object is an instance of a 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 Partition II) 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.

Note that:

1. Arrays inherit from `System.Array`
2. If Foo can be cast to Bar, then Foo[] can be cast to Bar[]
3. For the purposes of 2., enums are treated as their undertlying type: thus E1[] can cast to E2[] if E1 and E2 share an underlying type

Exceptions:

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

Verifiability:

Correct CIL ensures that *class* is a valid **`typeref`** or **`typedef`** token indicating a class, and that *obj* is always either null or an object reference

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

Format	Assembly Format	Description
90	ldelem.i1	Load the element with type **`int8`** at *index* onto the top of the stack as an **`int32`**
92	ldelem.i2	Load the element with type **`int16`** at *index* onto the top of the stack as an **`int32`**
94	ldelem.i4	Load the element with type **`int32`** at *index* onto the top of the stack as an **`int32`**
96	ldelem.i8	Load the element with type **`int64`** at *index* onto the top of the stack as an **`int64`**
91	ldelem.u1	Load the element with type **`unsigned int8`** at *index* onto the top of the stack as an **`int32`**
93	ldelem.u2	Load the element with type **`unsigned int16`** at *index* onto the top of the stack as an **`int32`**
95	ldelem.u4	Load the element with type **`unsigned int32`** at *index* onto the top of the stack as an **`int32`**
96	ldelem.u8	Load the element with type **`unsigned int64`** at *index* onto the top of the stack as an **`int64`** (alias for **`ldelem.i8`**)
98	ldelem.r4	Load the element with type **`float32`** at *index* onto the top of the stack as an **`F`**
99	ldelem.r8	Load the element with type **`float64`** at *index* onto the top of the stack as an **`F`**
97	ldelem.i	Load the element with type **`native int`** at *index* onto the top of the stack as an **`native int`**
9A	ldelem.ref	Load the element of type object, at *index* 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 **`int32`** or **`native int`**) 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 **`Get`** method.

Note that integer values of less than 4 bytes are extended to **`int32`** (not **`native int`**) 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 negative, or larger than the bound of *array*.

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

Verifiability:

Correct CIL code requires that *array* is either null or a zero-based, one-dimensional array whose declared element type matches exactly the type for this particular instruction suffix (e.g. **`ldelem.r4`** can only be applied to a zero-based, one dimensional array of **`float32`**s)

4.8 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 → …, address

Description:

The **ldelema** instruction loads the address of the element with index *index* (of type **int32** or **native int**) 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 address is a managed pointer (type **&**).

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

Exceptions:

`NullReferenceException` is thrown if *array* is null.

`IndexOutOfRangeException` is thrown if *index* is negative, or larger than the bound of *array*.

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

Verifiability:

Correct CIL ensures that *class* is a **typeref** or **typedef** token to a class, and that *array* is indeed always either null or a zero-based, one-dimensional array whose declared element type matches *class* exactly.

4.9 ldfld – load field of an object

Format	Assembly Format	Description
7B <*T*>	ldfld *field*	Push the value of *field* of object, or value type, *obj,* onto the stack

Stack Transition:

…, obj $\rightarrow$ …, 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 **`native int`**), or an instance of a value type. The use of an unmanaged pointer is not permitted in verifiable code. *field* is a metadata token (a **`fieldref`** or **`fielddef`** see Partition II) 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 CIL is converted to native code, not at runtime.

Verifiability:

Correct CIL ensures that *field* is a valid token referring to a 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.

4.10 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 of *obj*. *obj* is either an object, type **`O`**, a managed pointer, type **`&`**, or an unmanaged pointer, type **`native int`**. The use of an unmanaged pointer is not allowed in verifiable 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 **`native int`**).

field is a metadata token (a **`fieldref`** or **`fielddef;`** see Partition II) that must refer to a field member. The field may be either an instance field (in which case *obj* must not be null) or a static field.

Exceptions:

`InvalidOperationException` is thrown if the *obj* is not within the application domain from which it is being accessed. The address of a field that is not inside the accessing application domain cannot be loaded.

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

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

Verifiability:

Correct CIL 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 verification.

4.11 ldlen – load the length of an array

Format	Assembly Format	Description
8E	ldlen	Push the length (of type `native unsigned int`) of *array* on the stack

Stack Transition:

…, array → …, length

Description:

The `ldlen` instruction pushes the number of elements 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 `native unsigned int`.

Exceptions:

`NullReferenceException` is thrown if *array* is null.

Verifiability:

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

4.12 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, **`&`**, or unmanaged pointer, **`native unsigned int`**) 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 Partition II) representing a value type.

Rationale: *The **`ldobj`** instruction is used to pass a value type as a parameter. See Partition I.*

It is unspecified what happens if *addrOfValObj* 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 CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL ensures that *classTok* is a metadata token representing a value type and that *addrOfValObj* is a pointer to a location containing a value of the type specified by *classTok*. Verifiable code additionally requires that *addrOfValObj* is a managed pointer of a matching type.

4.13 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:

`…,` $\rightarrow$ `…, 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 Partition II) referring to a static field member. The return type is that associated with *field*.

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

Exceptions:

None.

Verifiability:

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

4.14 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 managed pointer, type **`&`**, if *field* refers to a type whose memory is managed; otherwise an unmanaged pointer, type **`native int`**) of a static field on the stack. *field* is a metadata token (a **`fieldref`** or **`fielddef;`** see Partition II) referring to a static field member. (Note that *field* may be a static global with assigned RVA, in which case its memory is *un*managed; where RVA stands for Relative Virtual Address, the offset of the field from the base address at which its containing PE file is loaded into memory)

Exceptions:

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

Verifiability:

Correct CIL ensures that *field* is a valid metadata token referring to a static field member if *field* refers to a type whose memory is managed.

Note: Using **`ldsflda`** 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 verification.

4.15 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* (that must be a string literal).

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 CLI guarantees that the result of two **ldstr** instructions referring to two metadata tokens that have the same sequence of characters return precisely the same string object (a process known as "string interning").

Exceptions:

None.

Verifiability:

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

4.16 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:

… → …, RuntimeHandle

Description:

The **ldtoken** instruction pushes a RuntimeHandle for the specified metadata token. The token must be one of:

A **methoddef** or **methodref** : pushes a **RuntimeMethodHandle**

A **typedef** or **typeref** : pushes a **RuntimeTypeHandle**

A **fielddef** or **fieldref** : pushes a **RuntimeFieldHandle**

The value pushed on the stack can be used in calls to Reflection methods in the system class library

Exceptions:

None.

Verifiability:

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

4.17 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 $\rightarrow$ …, ftn

Description:

The **`ldvirtftn`** instruction pushes an unmanaged pointer (type **`native int`**) 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 Partition II) 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 *mthd*. 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 CLI when a native version of the method isn't available)

Exceptions:

None.

Verifiability:

Correct CIL ensures that *mthd* is a valid **`methoddef`** or **`methodref`** token. Also that *mthd* references a non-static method that is defined for *object.* Verification tracks the type of the value pushed in more detail than the "**`native int`**" 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.

4.18 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 **`native int`**) that holds the address of a piece of data. *Class* is the class token (a **`typeref`** or **`typedef;`** see Partition II) describing the type of *ptr*. **`Mkrefany`** pushes a typed reference on the stack, that is 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.

Exceptions:

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

Verifiability:

Correct CIL ensures that *class* is a valid **`typeref`** or **`typedef`** token describing some type and that *ptr* is a pointer to exactly that type. Verification additionally requires that *ptr* be a managed pointer. Verification will fail if it cannot deduce that *ptr* is a pointer to an instance of *class*.

4.19 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 Partition II). *numElems* (of type native int) specifies the number of elements in the array. 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 array 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 Base Framework.

Exceptions:

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

`OverflowException` is thrown if *numElems* is < 0

Verifiability:

Correct CIL ensures that *etype* is a valid **`typeref`** or **`typedef`** token.

4.20 newobj – create a new object

Format	Assembly Format	Description
73 <*T*>	newobj *ctor*	Allocate an uninitialized object or value type 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 Partition II) 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, that 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 CIL is converted to native code, rather than at runtime.

Verifiability:

Correct CIL 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. Verification considers a delegate constructor as a special case, checking that the method pointer passed in as the second argument, of type **`native int,`** does indeed refer to a method of the correct type.

4.21 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:

Correct CIL ensures that *TypedRef* is a valid typed reference (created by a previous call to **`mkrefany`**). The **`refanytype`** instruction is always verifiable.

4.22 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 Partition II). See the **`mkrefany`** instruction.

Exceptions:

`InvalidCastException` is thrown if *type* is not identical to the type stored in the TypedRef (ie, the *class* supplied to the **`mkrefany`** instruction that constructed that TypedRef)

`TypeLoadException` is thrown if *type* cannot be found.

Verifiability:

Correct CIL ensures that *TypedRef* is a valid typed reference (created by a previous call to **`mkrefany`**). The **`refanyval`** instruction is always verifiable.

4.23 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 Partition I). It throws the same exception that was caught by this handler.

Exceptions:

The original exception is thrown.

Verifiability:

Correct CIL uses this instruction only within the body of a **catch** handler (not of any exception handlers embedded within that **catch** handler). If a rethrow occurs elsewhere, then an exception will be thrown, but precisely which exception is undefined

4.24 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 **`unsigned int32`**

Stack Transition:

…, $\rightarrow$ …, 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 Partition II) that specifies a value type.

Rationale: *The definition of a value type can change between the time the CIL is generated and the time that it is loaded for execution. Thus, the size of the type is not always known when the CIL is generated. The* ***`sizeof`*** *instruction allows CIL code to determine the size at runtime without the need to call into the Framework class library. The computation can occur entirely at runtime or at CIL-to-native-code compilation time.* ***`sizeof`*** *returns the total size that would be occupied by each element in an array of this value type – including any padding the implementation chooses to add. Specifically, array elements lie* ***`sizeof`*** *bytes apart*

Exceptions:

None.

Verifiability:

Correct CIL ensures that **`valueType`** is a **`typeref`** or **`typedef`** referring to a value type. It is always verificable.

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

Format	Assembly Format	Description
9C	stelem.i1	Replace array element at *index* with the **`int8`** value on the stack
9D	stelem.i2	Replace array element at *index* with the **`int16`** value on the stack
9E	stelem.i4	Replace array element at *index* with the **`int32`** value on the stack
9F	stelem.i8	Replace array element at *index* with the **`int64`** value on the stack
A0	stelem.r4	Replace array element at *index* with the **`float32`** value on the stack
A1	stelem.r8	Replace array element at *index* with the **`float64`** 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 **`int32`** or **`native int`**) 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 `ArrayTypeMismatchException`.

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 negative, or larger than the bound of *array*.

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

Verifiability:

Correct CIL requires that *array* be a zero-based, one-dimensional array whose declared element type matches exactly the type for this particular instruction suffix (eg **`stelem.r4`** can only be applied to a zero-based, one dimensional array of **`float32`**'s); also that *index* lies within the bounds of *array*

4.26 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 **`native int`**, or **`&`**) with **`value`**. **`field`** is a metadata token (a **`fieldref`** or **`fielddef;`** see Partition II) 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 CIL is converted to native code, not at runtime.

Verifiability:

Correct CIL ensures that *field* is a valid token referring to a 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 verification .

4.27 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 **`native int`**, 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 Partition II) 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 CIL is converted to native code rather than at runtime.

Verifiability:

Correct CIL 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 managed pointer.

4.28 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 Partition II) that must refer to a static field member. **Stsfld** pops the value off the stack and updates the static field with that value.

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 CIL is converted to native code, not at runtime.

Verifiability:

Correct CIL 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.

Note: Using **stsfld** 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 verification.

4.29 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 Partition I.

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

Exceptions:

`NullReferenceException` is thrown if *obj* is null.

Verifiability:

Correct CIL 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`.

4.30 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:

A value type has two separate representations (see Partition I) within the CLI:

- 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**) indicating the type of value type contained within *obj*. If *obj* is not a boxed instance of *valuetype*, or, if *obj* is a boxed enum and *valuetype* is not its underlying type, then this instruction will throw an InvalidCastException

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* (or if *obj* is a boxed enum and *valuetype* is not its underlying type)

NullReferenceException is thrown if obj is null.

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

Verifiability:

Correct CIL 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**, and represents a boxed instance of a *valuetype* value type.