Common Language Infrastructure (CLI)

Partition III:
CIL Instruction Set

III.1 Introduction

This partition is a detailed description of the Common Intermediate Language (CIL) instruction set, part of the specification of the CLI. 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.

In this partition, each instruction is described in its own subclause, one per page. Related CLI machine instructions are described together. Each instruction description consists of the following parts:

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

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

  `System.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 §III.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.]

  `System.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 (e.g., an implementation based on JIT compilation to native code can 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 §III.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 §III.1.5.

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

III.1.1 Data types

While the CTS defines a rich type system and the 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, collectively called the "basic CLI types", contains the following 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 can be assigned the value `null`. This is defined throughout the CLI to be zero (a bit pattern of all-bits-zero).

[*Note*: As far as VES operations on the evaluation stack are concerned, there is only one floating-point type, and the VES does not care about its size. The VES makes the distinction about the

size of numerical values only when storing these values to, or reading from, the heap, statics, local variables, or method arguments. *end note*]

III.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). However, the CIL instruction set 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) can hold 1- or 2-byte integers. For the purpose of stack operations the bool (§III.1.1.2) and char types are treated as unsigned 1-byte and 2-byte integers respectively. Loading from these locations onto the stack converts them to 4-byte values by:

  - zero-extending for types unsigned int8, unsigned int16, bool and char;
  - sign-extending for types int8 and int16;
  - zero-extends for unsigned indirect and element loads (ldind.u*, ldelem.u*, etc.); and
  - sign-extends for signed indirect and element loads (ldind.i*, ldelem.i*, etc.

  Storing to integers, booleans, and characters (stloc, stfld, stind.i1, 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 (i.e., 1- and 2-byte) integers are loaded as 4-byte numbers on all architectures and these 4-byte numbers are always 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 is executed) will have identical results on all implementations. *end note*]

  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 a short integer 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 cannot be directly combined with 8-byte integers (they shall 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 shall 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 can 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` shall 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 might be represented internally with additional range and/or precision. The size of the internal floating-point representation is implementation-dependent, might 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 (such as contractions), which result in the same or greater precision. Where exactly reproducible behavior precision is required by a language or application (e.g., the Kahan Summation Formula), explicit conversions can be used. Reproducible precision does not guarantee reproducible behavior, however. Implementations with extra precision might round twice: once for the floating-point operation, and once for the explicit conversion. Implementations without extra precision effectively round only once. In rare cases, rounding twice versus rounding once can yield results differing by one unit of least precision. *end note*]

  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 might involve a loss of precision or the creation of an out-of-range value (NaN, +infinity, or -infinity). However, the value might 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 shall 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. *end note*]

[*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. *end note*]

III.1.1.2 Boolean data type

A CLI Boolean type occupies 1 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. For the purpose of stack operations boolean values are treated as unsigned 1-byte integers (§III.1.1.1).

III.1.1.3 Character data type

A CLI char type occupies 2 bytes in memory and represents a Unicode code unit using UTF-16 encoding. For the purpose of stack operations char values are treated as unsigned 2-byte integers (§III.1.1.1).

III.1.1.4 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.

III.1.1.5 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 one from the other, producing a signed 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 shall 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 *shall* 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 behavior is unspecified.

III.1.1.5.1 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 valid 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., using 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.

III.1.1.5.2 Managed pointers (type &)

Managed pointers (`&`) can point to a local variable, a method argument, a field of an object, a field of a value type, an element of an array, a static field, 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 shall 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 byref 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

III.1.2 Instruction variant table

In §III.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 operands 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 show how this instruction is encoded (its "opcode"). For example, the ldarga instruction is encoded as a byte holding FE, followed by another holding 0A. Italicized type names delimited by < and > represent numbers that should follow in the instruction stream; for example, a 2-byte quantity that is to be treated as an unsigned integer directly follows the FE 0A opcode. [*Example*: One of the forms of the ldc.<*type*> instruction is ldc.r8 num, which has a Format "23 <*float64*>". For the instruction ldc.r8 3.1415926535897931, the resulting code is 23 182D4454FB210940, where 182D4454FB210940 is the 8-byte hex representation for 3.1415926535897931.

Similarly, another of the forms of the ldc.<*type*> instruction is ldc.i4.s *num*, which a Format of "1F <*int8*>". For the instruction ldc.i4.s -3, the resulting code is 1F FD, where FD is the 1-byte hex representation for -3. The .s suffix indicates an instruction is a short-form instruction. In this case, it requires 2 bytes rather than the long form ldc.i4, which requires 5 bytes. *end example*]

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 operand 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 operand numbers are encoded least-significant-byte-at-smallest-address (a pattern commonly termed "little-endian"). Bytes for instruction opcodes and operands 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 having instruction stream operands, this column also assigns names to each of the operands to the instruction. For each instruction operand, there is a name in the assembly format. These names are used later in the instruction description.

III.1.2.1 Opcode encodings

CIL opcodes are one or more bytes long; they can 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 III.1: Opcode Encodings

Opcode	Instruction
0x00	nop
0x01	break

Opcode	Instruction
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

Opcode	Instruction
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

Opcode	Instruction
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

Opcode	Instruction
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

Opcode	Instruction
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
0xA3	ldelem
0xA4	stelem
0xA5	unbox.any
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

Opcode	Instruction
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 0x16	constrained.
0xFE 0x17	cpblk
0xFE 0x18	initblk
0xFE 0x19	no.
0xFE 0x1A	rethrow
0xFE 0x1C	sizeof
0xFE 0x1D	Refanytype
0xFE 0x1E	readonly.

III.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 shall 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, across the page). The instruction removes these values from the stack and replaces them by another value, called *result* in the description.

III.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.

III.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 `&`.

Table III.2 shows the result type for A `op` B—where op is add, div, mul, rem, or sub—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 III.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	×	×	×	×	×	×

Table III.3 shows the result type for the unary numeric operations. Used for the neg instruction. Boxes marked × indicate an invalid CIL instruction. All valid uses of this instruction are verifiable.

Table III.3: Unary Numeric Operations

Operand Type	int32	int64	native int	F	&	O
Result Type	int32	int64	native int	F	×	×

Table III.4 shows the result type for the comparison and branch instructions. The binary comparison returns a Boolean value and the branch operations 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 III.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, beq.s, bne.un.s, 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 *end note*]

[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)

Table III.5 shows the result type for each possible combination of operand types in integer operations. Used for and, div.un, not, or, rem.un, xor. The div.un and rem.un instructions treat their operands as unsigned integers and produce the bit pattern corresponding to the unsigned result. As described in the CLI standard, 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 shall be of type int32 or native int. **Boxes** marked × indicate invalid CIL sequences. All other boxes denote verifiable combinations of operands.

Table III.5: Integer Operations

	int32	int64	native int	F	&	O
int32	int32	×	native int	×	×	×
int64	×	int64	×	×	×	×
native int	native int	×	native int	×	×	×
F	×	×	×	×	×	×
&	×	×	×	×	×	×
O	×	×	×	×	×	×

Table III.6 shows the valid 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 unspecified. (e.g., shift an int32 integer left by 37 bits)

Table III.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	×	×	×	×	×	×

Table III.7 shows the result type for each possible combination of operand types in the arithmetic operations with overflow checking. An exception shall be thrown if the result cannot be represented in the result type. Used for add.ovf, add.ovf.un, mul.ovf, mul.ovf.un, sub.ovf, and sub.ovf.un. For details of the exceptions thrown, see the descriptions of the specific instructions. The shaded uses are not verifiable, while **boxes** marked × indicate invalid CIL sequences.

Table III.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	×	×	×	×	×	×

Table III.8 shows the result type for the conversion operations. Conversion operations convert the top item on the evaluation stack from one numeric type to another. While converting, truncation or extension occurs as shown in the table. 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 an 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 III.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 (i.e., 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 result type is unsigned) or sign-extended (if the result type is signed). Thus, converting the value 0x1234 ABCD from the evaluation stack to an 8-bit datum yields the result 0xCD; if the result type were `int8`, this is sign-extended to give 0xFFFF FFCD; if, instead, the result type were `unsigned int8`, this is zero-extended to give 0x0000 00CD.

[2] "Truncate to zero" 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).

Rounding mode for integer to and from F conversions is the same as for arithmetic.

III.1.6 Implicit argument coercion

A method call involves the implicit assignment of argument values on the stack to the parameters of the called method (accessed using the `ldarg`, §III.3.38, or `ldarga`, §III.3.39, instructions). The assignment is an implicit `starg` (§III.3.61) instruction and may be referred to as *implicit argument coercion*.

In Verified CLI the validity of implicit argument coercion, as with the `starg` (§III.3.61) instruction, is determined by the *verifier-assignable-to* relation (§III.1.8.1.2.3). Correct CIL also allows a `native int` to be passed as a byref (`&`); in which case following implicit conversion the value will be tracked by garbage collection.

The remainder of this clause contains only informative text

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 might 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 III.9: Signature Matching

Type In Signature	**Stack Parameter**						
	`int32`	`native int`	`int64`	`F`	`&`	`O`	`value type` (Note[1])
`int8`	✓ **Truncate**	✓ **Truncate**	×	×	×	×	×
`unsigned int8, bool`	✓ **Truncate**	✓ **Truncate**	×	×	×	×	×

`int16`	✓ **Truncate**	✓ **Truncate**	×	×	×	×	×
`unsigned int16, char`	✓ **Truncate**	✓ **Truncate**	×	×	×	×	×
`int32`	✓ **Nop**	✓ **Truncate**	×	×	×	×	×
`unsigned int32`	✓ **Nop**	✓ **Truncate**	×	×	×	×	×
`int64`	×	×	✓ **Nop**	×	×	×	×
`unsigned int64`	×	×	✓ **Nop**	×	×	×	×
`native int`	✓ Sign extend	✓ **Nop**	×	×	×	×	×
`native unsigned int`	✓ Zero extend	✓ Nop	×	×	×	×	×
`float32`	×	×	×	**Note**[4]	×	×	×
`float64`	×	×	×	**Note**[4]	×	×	×
Class	×	×	×	×	×	✓	×
Value Type (Note[1])	×	×	×	×	×	×	✓ (Note[2])
By-reference (Byref) (`&`)	×	✓ Start GC tracking	×	×	✓	×	×
Typed Reference (RefAny) (Note[3])	×	×	×	×	×	×	×

[1] A value type in a signature cannot be the long form of a built-in type (§II.23.2.15).

[2] The CLI's stack can contain a value type. These can only be passed if the particular value type on the stack exactly matches the value type required by the corresponding parameter.

[3] There are special instructions to construct and pass a `RefAny`.

[4] The CLI is permitted to pass floating point arguments using its internal F type, see §III.1.1.1. CIL generators can, of course, include an explicit conv.r4, conv.r4.ovf, or similar instruction.

Further notes concerning this table:

- The meaning of Truncate is defined for Table 8 above; Nop means no conversion is performed.
- "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.

III.1.7 Restrictions on CIL code sequences

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

- Correct CIL

- Verifiable CIL

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

III.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, CIL 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 can 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 can appear as the result of machine-generated code and is allowed, but it shall 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. *end note*]

III.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".]

III.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 §III.1.7.2. It is invalid for a protected block, filter clause, or handler to end without forming a complete last instruction.

III.1.7.4 Must provide maxstack

Every method specifies a maximum number of items that can be pushed onto the CIL evaluation stack. 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 shall be tracked by an analysis tool. *end note*]

[*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. *end rationale*]

III.1.7.5 Backward branch constraints

It shall 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. *end rationale*]

[*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. *end note*]

See the following for further information:

- Exceptions: Partition I
- Verification conditions for branch instructions: §III.3
- The tail. prefix: §III.3.19

III.1.7.6 Branch verification constraints

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

III.1.8 Verifiability and correctness

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.

Correct CIL is CIL that executes on all conforming implementations of the CLI, with well-defined behavior as specified in this standard. However, correct CIL need not result in identical behavior across conforming implementations; that is, the behavior might be implementation-specific.

It is perfectly acceptable to generate correct CIL code that is not verifiable, but which is known to be memory safe by the compiler writer. Thus, correct CIL might not be verifiable, even though the producing compiler might *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.

Correct 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 which are not verifiable, might violate type safety and hence might violate security or memory access constraints. See II.3 for additional information.

III.1.8.1 Flow control restrictions for verifiable CIL

This subclause specifies a verification algorithm that, combined with information on individual CIL instructions (see §III.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 might violate memory safety. Any program that is verifiable according to this standard does not violate memory safety, and a conforming implementation of the CLI shall run such programs. The implementation might 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 might 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.

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. *end note*]

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 §III.3), and on the overall file format in Partition II.

III.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 shall 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 localsinit bit set, see Partition II (Flags for Method Headers). If this bit is not set, then a CLI might 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 CIL-to-native-code compiler to minimize its performance impact. Note that a CLI might optionally choose to perform definite-assignment analysis – in such a case, it might confirm that a method, even without the localsinit bit set, might in fact be verifiable (and therefore not throw a Verification exception) *end rationale*]

[*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 VES.

Performance measurements on C++ implementations (which do 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. *end note*]

The verification algorithm shall simulate all possible control flow paths through the code and ensure that a valid 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 §III.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.

With the exception of the unconditional branch instructions, **throw**, **rethrow**, and **ret**, there is a control flow path from every instruction to the subsequent instruction. There is also a control flow path from each branch instruction (conditional or unconditional) to the branch target (or targets, in the case of 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 §III.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 might be more than one such address) that cannot be merged with the stack state just computed. For rules of this merge operation, see §III.1.8.1.3.

The CLI supports the notion of a *controlled-mutability* managed pointer. (See §III.1.8.1.2.2, the merging rules in §III.1.8.1.3, the **readonly.** instruction prefix in §III.2.3, the **ldfld** instruction in §III.4.10, the **stfld** instruction in §III.4.30, and the **unbox** instruction in §III.4.32.)

The VES ensures that both special constraints and type constraints are satisfied. The constraints can be checked as early as when a closed type is constructed, or as late as when a method on the constrained generic type is invoked, a constrained generic method is invoked, a field in a constrained generic type is accessed, or an instance of a constrained generic type is created.

To accommodate generics, the type compatibility relation is extended to deal with:

- generic parameters: a generic parameter is only *assignable-to* (§I.8.7.3) itself.
- boxed generic parameters: a boxed generic parameter is *assignable-to* (§I.8.7.3)the constraint types declared on the generic parameter.

In the verification semantics, boxing a value of primitive or value type on the stack introduces a value of type "boxed" type; if the value type is Nullable<T> (Partition I.8.2.4), a value of type "boxed" *T* is introduced. This notion of boxed type is extended to generic parameters. Boxing a value whose type is a generic parameter (!0, for example) introduces a value of the boxed parameter type on the stack ("boxed" !0, for example). The boxed forms of value types, and now generic parameters, are used to support efficient instance and virtual method calls on boxed values. Because the "boxed" type statically records the exact type of the underlying value, there is no need to perform a checked cast on the instance from some less informative, but syntactically expressible, reference type.

Just like the boxed forms of primitive and non-primitive value types, the boxed forms of generic parameters only occur on the verification stack (after being introduced by a `box` instruction). They cannot be explicitly specified using metadata signatures.

III.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 §III.1.8.1.2.1, the type compatibility rules are specified in §III.1.8.1.2.2, and the rules for merging stack states are in §III.1.8.1.3.

III.1.8.1.2.1 Verification types

§I.8.7 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. A null value may be known to have some reference type; e.g., when it has been loaded from a local or field; or to have the special null type when it results from a ldnull instruction. A null value of null type can only exist on the evaluation stack. When the correctness or verification sections (§III.1.8) of any instruction require a value of some particular reference type, then a value of null type is also permitted. If a value of null type is supplied and the instruction dereferences it, then a System.NullReferenceException is thrown; this is noted in the appropriate exception areas of the instruction descriptions.

This block contains only informative text.

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 §I.8.7, *verification type*, and §III.1.8.1.2.2) when considering the types of local variables, arguments, and parameters on methods being called. The column "Verification Type (in stack state)" corresponds with *intermediate type*, §I.8.7, and 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 (see §I.8.7.2, *pointer-element-compatible-with*).

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

End informative text

A method can be defined as returning a managed pointer, but calls upon such methods are not verifiable. When returning byrefs, verification is done at the return site, not at the call site.

[*Rationale:* Some uses of returning a managed pointer are perfectly verifiable (e.g., returning a reference to a field in an object); but some not (e.g., 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. *end rationale*]

III.1.8.1.2.2 Controlled-mutability managed pointers

The readonly. prefix and unbox instructions can produce what is called a *controlled-mutability managed pointer*. Unlike ordinary managed pointer types, a controlled-mutability managed pointer is not *verifier-assignable-to* (§III.1.8.1.2.3) ordinary managed pointers; e.g., it cannot be passed as a byref argument to a method. At control flow points, a controlled-mutability managed pointer can be merged with a managed pointer of the same type to yield a controlled-mutability managed pointer.

Controlled-mutability managed pointers can only be used in the following ways:

1. As the object parameter for an ldfld, ldflda, stfld, call, callvirt, or constrained. callvirt instruction.
2. As the pointer parameter to a ldind.* or ldobj instruction.
3. As the source parameter to a cpobj instruction.

All other operations (including stobj, stind.*, initobj, and mkrefany) are invalid.

The pointer is called a controlled-mutability managed pointer because the defining type decides whether the value can be mutated. For value classes that expose no public fields or methods that update the value in place, the pointer is read-only (hence the name of the prefix). In particular, the classes representing primitive types (such as `System.Int32`) do not expose mutators and thus are read-only.

III.1.8.1.2.3 Verification type compatibility

Verification type compatibility is defined in terms of assignment compatibility (see §I.8.7).

A type Q is *verifier-assignable-to* R (sometimes written R := Q) if and only if T is the verification type of Q, and U is the verification type of R, and at least one of the following holds:

1. T is identical to U. [Note: this is reflexivity for verification type compatibility.]
2. There exists some V such that T is *verifier-assignable-to* V and V is *verifier-assignable-to* U. [Note: this is transitivity for verification type compatibility.]
3. T is *assignable-to* U according to the rules in §I.8.7.3.
4. T is a controlled-mutability managed pointer type to type V and U is a controlled-mutability managed pointer type to type W and V is *pointer-element-assignable-to* W.
5. T is a managed pointer type V& and U is a controlled-mutability managed pointer type to type W and V is *pointer-element-assignable-to* W.
6. T is boxed V and U is the immediate base class of V.
7. T is boxed V and U is an interface directly implemented by V.
8. T is boxed X for a generic parameter X and V is a generic constraint declared on parameter X.
9. T is the null type, and U is a reference type.

[*Note*: *verifier-assignable-to* extends *assignable-to* to deal with types that can occur only on the stack, namely boxed types, controlled-mutability managed pointer types, and the null type. *end note*]

In the remainder of Partition III, the use of the notation "U := T" is sometimes used to mean T is *verifier-assignable-to* U.

III.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 §III.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.

Merging a controlled-mutability managed pointer with an ordinary (that is, non-controlled-mutability) managed pointer to the same type results in a controlled-mutability managed pointer to that type.

III.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. *end note*]

III.1.8.1.5 Delegate construction

Verification of delegate construction is based on code sequences rather than individual instructions. These are detailed in the description of the newobj instruction (§III.4.21).

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. *end note*]

III.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)

III.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.

III.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.

III.2.1 constrained. – (prefix) invoke a member on a value of a variable type

Format	Assembly Format	Description
FE 16 <*T*>	constrained. *thisType*	Call a virtual method on a type constrained to be type T

Stack Transition:

…, ptr, arg1, … argN → …, ptr, arg1, … argN

Description:

The `constrained.` prefix is permitted only on a callvirt instruction. The type of *ptr* must be a managed pointer (&) to *thisType*. The constrained prefix is designed to allow callvirt instructions to be made in a uniform way independent of whether *thisType* is a value type or a reference type.

When callvirt *method* instruction has been prefixed by constrained *thisType* the instruction is executed as follows.

If *thisType* is a reference type (as opposed to a value type) then

ptr is dereferenced and passed as the 'this' pointer to the callvirt of *method*

If *thisType* is a value type and *thisType* implements *method* then

ptr is passed unmodified as the 'this' pointer to a call of *method* implemented by *thisType*

If *thisType* is a value type and *thisType* does not implement *method* then

ptr is dereferenced, boxed, and passed as the 'this' pointer to the callvirt of *method*

This last case can only occur when *method* was defined on `System.Object`, `System.ValueType`, or `System.Enum` and not overridden by *thisType*. In this last case, the boxing causes a copy of the original object to be made, however since all methods on `System.Object`, `System.ValueType`, and `System.Enum` do not modify the state of the object, this fact cannot be detected.

The need for the constrained prefix was motivated by the needs IL generators creating generic code. Normally the callvirt instruction is not valid on value types. Instead it is required that IL compilers effectively perform the `this' transformation outlined above at IL compile time, depending on the type of ptr and the method being called. It is not possible to do this transformation at IL compile time, however, when ptr is a generic type (which is unknown at IL compile time). This is why the constrained prefix is needed. The constrained opcode allows IL compilers to make a call to a virtual function in a uniform way independent of whether ptr is a value type or reference type. While this was targeted for the case were *thisType* is a generic type variable, constrained works for non-generic types too, and can ease the complexity of generating virtual calls in languages that hide the distinction between value and reference types.

Exceptions:

None.

Correctness:

The constrained prefix will be immediately followed by a callvirt instruction. *thisType* shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

Verifiability:

The ptr argument will be a managed pointer (&) to *thisType*. In addition all the normal verification rules of the callvirt instruction apply after the ptr transformation as described above. This is equivalent to requiring that a boxed *thisType* must be a subclass of the class which *method* belongs to.

[*Rationale:* The goal of this instruction was to achieve uniformity of calling virtual functions, so such calls could be made verifiably in generic routines. One way of achieving this uniformity was to always box the 'this' pointer before making a callvirt. This works for both reference type (where box is a no-op), and value types. The problem with this approach is that a copy is made in the value type case. Thus if the method being called modifies the state of the value type, this will not be reflected after the call completes since this modification was made in the boxed copy. This semantic difference (as well as the performance cost of the extra boxing), makes this alternative unacceptable. *end rationale*]

III.2.2 no. – (prefix) possibly skip a fault check

Format	Assembly Format	Description
FE 19 <unsigned int8>	no. { typecheck | rangecheck | nullcheck }	The specified fault check(s) normally performed as part of the execution of the subsequent instruction can/shall be skipped.

Description:

This prefix indicates that the subsequent instruction need not perform the specified fault check when it is executed. The byte that follows the instruction code indicates which checks can optionally be skipped. This instruction is not verifiable.

The prefix can be used in the following circumstances:

0x01: typecheck (castclass, unbox, ldelema, stelem, stelem). The CLI can optionally skip any type checks normally performed as part of the execution of the subsequent instruction. InvalidCastException can optionally still be thrown if the check would fail.

0x02: rangecheck (ldelem.*, ldelema, stelem.*). The CLI can optionally skip any array range checks normally performed as part of the execution of the subsequent instruction. IndexOutOfRangeException can optionally still be thrown if the check would fail.

0x04: nullcheck (ldfld, stfld, callvirt, ldvirtftn, ldelem.*, stelem.*, ldelema). The CLI can optionally skip any null-reference checks normally performed as part of the execution of the subsequent instruction. NullReferenceException can optionally still be thrown if the check would fail.

The byte values can be OR-ed; e.g.; a value of 0x05 indicates that both typecheck and nullcheck can optionally be omitted.

Exceptions:

None.

Correctness:

Correct IL permits the prefix only on the instructions specified above.

Verifiability:

Verifiable IL does not permit the use of no.

III.2.3 readonly. (prefix) – following instruction returns a controlled-mutability managed pointer

Format	Assembly Format	Description
FE 1E	readonly.	Specify that the subsequent array address operation performs no type check at runtime, and that it returns a controlled-mutability managed pointer

Description:

This prefix can only appear only immediately preceding the ldelema instruction and calls to the special *Address* method on arrays. Its effect on the subsequent operation is twofold.

1. At run-time, no type check operation is performed. (For the value class case there is never a runtime time check so this is a noop in that case).

2. The verifier treats the result of the address-of operation as a controlled-mutability managed pointer (§III.1.8.1.2.2).

Exceptions:

None.

Correctness:

Verifiability:

A controlled-mutability managed pointer must obey the verifier rules given in (2) of §III.1.8.1.2.2. See also §III.1.8.1.3.

[*Rationale:* The main goal of the readonly. prefix is to avoid a type check when fetching an element from an array in generic code. For example the expression

array[*i*].*method*()

where *array* has type T[] (where T is a generic parameter), and T has been constrained to have an interface with method '*method*' might compile into the following IL code.

```
ldloc array
ldloc j         // j is array index
readonly.
ldelema !0            // loads the pointer to the object
…               // load the arguments to the call
constrained. !0
callvirt method
```

Without the readonly. prefix the ldelema would do a type check in the case that `!0` was a reference class. Not only is this type check inefficient, but it is semantically incorrect. The type check for ldelema does an exact match typecheck, which is too strong in general. If the array held derived classes of `!0` then the code above would fail the ldelema typecheck. The only reason we fetch the address of the array element instead of the element itself (which is what the source code says), is because we need a handle for *array[i]* that works both for value types and reference types that can be passed to the constrained callvirt instruction.

If the array holds elements of a reference type, in general, skipping the runtime check would be unsafe. To be safe we have to insure that no modifications of the array happen through this pointer. The verifier rules stated above insure this. Since we explicitly allow read-only pointers to be passed as the object of instance method calls, these pointers are not strictly read-only for value types, but there is no type safety problem for value types. *end rationale*]

III.2.4 tail. (prefix) – call terminates current method

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

Description:

The tail. prefix shall 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 shall be empty except for the arguments being transferred by the following call. The instruction following the call instruction shall be a ret. Thus the only valid code sequence is

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

Correct CIL shall not branch to the call instruction, but it is permitted to branch to the ret. The only values on the stack shall 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 can 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 can 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. *end rationale*]

Exceptions:

None.

Correctness:

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 shall be *assignable-to* (§I.8.7.3)the return type of the current method.

Verifiability:

Verification requires that no managed pointers are passed to the method being called, since it does not track pointers into the current frame.

III.2.5 unaligned. (prefix) – pointer instruction might be unaligned

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

Stack Transition:

…, addr → …, addr

Description:

The unaligned. prefix specifies that *addr* (an unmanaged pointer (`&`), or `native int`) on the stack mignt 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* might 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) shall 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. *end rationale*]

The unaligned. and volatile. prefixes can be combined in either order. They shall immediately precede a ldind, stind, ldfld, stfld, ldobj, stobj, initblk, or cpblk instruction.

[*Note:* See Partition I, 12.7 for information about atomicity and data alignment. *end note*]

Exceptions:

None.

Correctness and Verifiability:

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

III.2.6 volatile. (prefix) – pointer reference is volatile

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

Stack Transition:

…, addr → …, addr

Description:

The volatile. prefix specifies that *addr* is a volatile address (i.e., it can 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 shall be marked separately. Access to volatile locations need not be performed atomically. (See Partition I, "Memory Model and Optimizations")

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

Exceptions:

None.

Correctness and Verifiability:

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

III.3 Base instructions

These instructions form a “Turing Complete” set of basic operations. They are independent of the object model that might be employed. Operations that are specifically related to the CTS’s object model are contained in the Object Model Instructions section.

III.3.1 add – add numeric values

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

Stack Transition:

…, value1, value2 → …, result

Description:

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

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

Exceptions:

None.

Correctness and Verifiability:

See Table 2: Binary Numeric Operations.

III.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 are encapsulated in Table 7: Overflow Arithmetic Operations.

Exceptions:

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

Correctness and Verifiability:

See Table 7: Overflow Arithmetic Operations.

III.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 are encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table 5: Integer Operations.

III.3.4 arglist – get argument list

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

Stack Transition:

… → …, argListHandle

Description:

The arglist instruction returns an opaque handle (having type `System.RuntimeArgumentHandle`) 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 can 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`. *end rationale*]

Exceptions:

None.

Correctness:

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.

Verifiability:

Its use is verifiable within the body of a method whose signature indicates it accepts a variable number of arguments, but verification requires that the result be an instance of the `System.RuntimeArgumentHandle` class.

III.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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.3.7 bge.un.<length> – branch on greater than or equal to, 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 floating-point values).

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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 floating-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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.3.11 ble.un.<length> – branch on less than or equal to, 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 floating-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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 shall use the **leave** instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 floating-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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 floating-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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 shall 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 might 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 valid. *end rationale*]

Exceptions:

None.

Correctness:

Correct CIL shall observe all of the control transfer rules specified above.

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 might trap to a debugger, do nothing, or raise a security exception: the exact behavior is implementation-defined.

Exceptions:

None.

Correctness:

Verifiability:

The break instruction is always verifiable.

III.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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

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

Verifiability:

Verifiable code requires the type-consistency of the stack, locals and arguments for every possible path to the destination instruction. See §III.1.8 for more details.

III.3.19 call – call a method

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

Stack Transition:

…, arg0, arg1 … argN → …, retVal (not always returned)

Description:

The call instruction calls the method indicated by the descriptor *method*. *method* is a metadata token (a `methodref`, `methoddef`, or `methodspec`;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 can be immediately preceded by a tail. prefix to specify that the current method state should be released before transferring control (see §III.2.3).

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 this with the callvirt instruction for calling virtual methods, where the destination address also depends upon the exact type of the instance reference pushed before the callvirt; see below.)

The CLI resolves the method to be called according to the rules specified in §I.12.4.1.3 (Computed destinations), except that the destination is computed with respect to the class specified by the metadata token.

[*Rationale:* This implements "call base class" behavior. *end rationale*]

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, and so on. There are three important special cases:

1. Calls to an instance (or virtual, see below) method shall push that instance reference (the `this` pointer) first. The signature carried in the metadata may not contain an entry in the parameter list for the `this` pointer but the calling convention always indicates whether one is required and if its signature is explicit or inferred (see §I.8.6.1.5 and §II.15.3) [*Note:* for calls to methods on value types, the `this` pointer is a managed pointer, not an instance reference §I.8.6.1.5. *end note*]
2. It is valid 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 can be called with either the call or callvirt instruction.

The arguments are passed as though by implicit `starg` (§III.3.61) instructions, see *Implicit argument coercion* §III.1.6.

call pops the `this` pointer, if any, 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 *arg0* parameter/`this` pointer is accessed as argument 0, *arg1* as argument 1, and so on.

Exceptions:

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

`System.MethodAccessException` can be thrown when there is an invalid attempt to access a non-public method.

`System.MissingMethodException` can be thrown when there is an attempt to dynamically access a method that does not exist.

Correctness:

Correct CIL ensures that the stack contains a `this` pointer if required and the correct number and type of arguments for the method being called. Unlike Verified CIL, Correct CIL also allows a `native int` to be passed as a byref (`&`); in which case following the store the value will be tracked by garbage collection.

Verifiability:

For a typical use of the **call** instruction, verification checks that:

(a) *method* refers to a valid `methodref`, `methoddef`, or `methodspec` token;

(b) if *method* requires a `this` pointer, as specified by its method signature (§I.8.6.1.5), then one is one the stack and its *verification type* is *verifier-assignable-to* (§III.1.8.1.2.3) the `this` signature of the method's signature;

(c) the types of the arguments on the stack are *verifier-assignable-to* (§III.1.8.1.2.3) the parameter signatures of the method's signature;

(d) the method is accessible from the call site; and

(e) the method is not abstract (i.e., it has an implementation).

If the call returns a value then verification also tracks that the type of the value returned as the *intermediate type* of the called method's return type.

The **call** instruction can also be used to call an object's base class 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 shall be static.

When using the **call** opcode to call a non-final virtual method on an instance other than a boxed value type, verification checks that the instance reference to the method being called is the result of **ldarg.s 0**, **ldarg 0** or **ldarg.0** and the caller's body does not contain **starg.s 0**, **starg 0** or **ldarga.s 0**, **ldarga 0**.

[*Rationale*: This means that non-virtually calling a non-final virtual method is only verifiable in the case where the subclass methods calls one of its superclasses using the same `this` object reference, where "same" is easy to verify. This means that an override implementation effectively "hides" the superclass' implementation, and can assume that the override implementation cannot be bypassed by code outside the class hierarchy.

For non-sealed class hierarchies, malicious code can attempt to extend the class hierarchy in an attempt to bypass a class' override implementation. However, this can only be done on an object of the malicious type, and not of the class with the override, which mitigates much of the security concern. *end rationale*]

III.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:

…, arg0, arg1 … argN, ftn → …, retVal (not always returned)

Description:

The calli instruction calls *ftn* (a pointer to a method entry point) with the arguments `arg0` … `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 can 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 originating 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 must be a method pointer to a method 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 could 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, and so on. The argument-building code sequence for an instance or virtual method shall push that instance reference (the `this` pointer, which shall not be `null`) first. [*Note:* for calls to methods on value types, the `this` pointer is a managed pointer, not an instance reference. §I.8.6.1.5. *end note*]

The arguments are passed as though by implicit `starg` (§III.3.61) instructions, see *Implicit argument coercion* §III.1.6.

calli pops the `this` pointer, if any, 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 *arg0* parameter/`this` pointer is accessed as argument 0, *arg1* as argument 1, and so on.

Exceptions:

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

Correctness:

Correct CIL requires that the function pointer contains the address of a method whose signature is *method-signature compatible-with* that specified by *callsitedescr* and that the arguments correctly correspond to the types of the destination function's `this` pointer, if required, and parameters. For the purposes of signature matching, the `HASTHIS` and `EXPLICITTHIS` flags are ignored; all other items must be identical in the two signatures. Unlike Verified CIL, Correct CIL also allows a `native int` to be passed as a byref (`&`); in which case following the store the value will be tracked by garbage collection.

[*Note*: In correct CIL, the required type of an instance function's *this* pointer is not included in *callsitedescr* if `HASTHIS` is set and `EXPLICITTHIS` is not set; but to be correct, the type of the supplied this parameter must be appropriate for the called function. *end note*]

Verifiability:

Verification checks that:

(a) the tracked type of *ftn* is a method signature (e.g., *ftn* was generated by ldftn, ldvirtfn, or loaded from a variable with the function type);

(b) if *ftn's* tracekd method signature specifies an instance method then a value for this pointer is on the stack and its *verification type* is *verifiable-assignable-to* (§III.1.8.1.2.3) method signature's `this` pointer; and

(c) the argument types are *verifier-assignable-to* (§III.1.8.1.2.3) the types of *ftn's* tracked method signature parameters.

If the call returns a value then verification also tracks that the type of the value returned as the *intermediate type* of *ftn's* tracked method signature's return type.

[*Note*: In the case of calling via a method pointer produced by `ldvirtftn`, which has a statically indeterminate *this* pointer type (and thus did not verify), the `calli` instruction does not verify. *end note*]

[*Note*: Verification is based on the tracked type of *ftn* and not *callsitedescr* as the former may carry the type of *this* in the case that the latter does not. However, verification requires correctness so the tracked type of *ftn* must be *method-signature compatible-with sitedescr*, the latter is not simply ignored. *end note*]

III.3.21 ceq – compare equal

Format	Assembly Format	Description
FE 01	Ceq	Push 1 (of type int32) if *value1* equals *value2*, else push 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

There are no additional verification requirements.

III.3.22 cgt – compare greater than

Format	Assembly Format	Description
FE 02	Cgt	Push 1 (of type int32) if *value1* > *value2*, else push 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 with 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

There are no additional verification requirements.

III.3.23 cgt.un – compare greater than, unsigned or unordered

Format	Assembly Format	Description
FE 03	cgt.un	Push 1 (of type int32) if *value1* > *value2*, unsigned or unordered, else push 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

There are no additional verification requirements.

III.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 → …, 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 behavior is unspecified if *value* is not a floating-point number.

Exceptions:

`System.ArithmeticException` is thrown if *value* is a NaN or an infinity.

Correctness:

Correct CIL guarantees that *value* is a floating-point number.

Verifiability:

There are no additional verification requirements.

III.3.25 clt – compare less than

Format	Assembly Format	Description
FE 04	Clt	Push 1 (of type int32) if *value1* < *value2*, else push 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

There are no additional verification requirements.

III.3.26 clt.un – compare less than, unsigned or unordered

Format	Assembly Format	Description
FE 05	clt.un	Push 1 (of type int32) if *value1* < *value2*, unsigned or unordered, else push 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.

Correctness:

Correct CIL provides two values on the stack whose types match those specified in Table 4: Binary Comparison or Branch Operations

Verifiability:

There are no additional verification requirements.

III.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. The verification type on the stack is as specified in §III.1.8.1.2.1 for the target type. 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 might 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 when 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, or if the floating-point value being converted to an integer is a NaN, the value returned is unspecified. The conv.r.un operation takes an integer off the stack, interprets it as unsigned, and replaces it with an F type floating-point number to represent the integer.

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

Exceptions:

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.

Correctness:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack.

Verifiability:

The table Table 8: Conversion Operations specifies a restricted set of types that are acceptable in verified code.

III.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 an unsigned int8 (on the stack as int32) and throw an exception on overflow.
B6	conv.ovf.u2	Convert to an unsigned int16 (on the stack as int32) and throw an exception on overflow.
B8	conv.ovf.u4	Convert to an unsigned int32 (on the stack as int32) and throw an exception on overflow
BA	conv.ovf.u8	Convert to an unsigned int64 (on the stack as int64) and throw an exception on overflow.
D4	conv.ovf.i	Convert to a 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 result cannot be represented in 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:

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

Correctness:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack.

Verifiability:

The table Table 8: Conversion Operations specifies a restricted set of types that are acceptable in verified code.

III.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 on the top of the stack is treated as an unsigned value before the conversion.

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 are encapsulated in Table 8: Conversion Operations.

Exceptions:

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

Correctness:

Correct CIL has at least one value, of a type specified in Table 8: Conversion Operations, on the stack.

Verifiability:

The table Table 8: Conversion Operations specifies a restricted set of types that are acceptable in verified code.

III.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 operation of the cpblk instruction can be altered by an immediately preceding volatile. or unaligned. prefix instruction.

[*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. *end rationale*]

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

Correctness:

CIL ensures the conditions specified above.

Verifiability:

The cpblk instruction is never verifiable.

III.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 are encapsulated in Table 2: Binary Numeric Operations.

Exceptions:

Integral operations throw `System.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
```

Correctness and Verifiability

See Table 2: Binary Numeric Operations.

III.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:

System.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
```

Correctness and Verifiability

See Table 5: Integer Operations.

III.3.33 dup – duplicate the top value of the stack

Format	Assembly Format	Description
25	Dup	Duplicate the 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.

Correctness and Verifiability:

No additional requirements.

III.3.34 endfilter – end exception handling filter clause

Format	Assembly Format	Description
FE 11	Endfilter	End an exception handling filter clause.

Stack Transition:

…, value → …

Description:

Used to return from the filter clause of an exception (see the Exception Handling subclause of Partition I for a discussion of exceptions). *value* (which shall be of type `int32` and 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.

The result of using any other integer value is unspecified.

The entry point of a filter, as shown in the method's exception table, shall be the (lexically) first instruction in the filter's code block. The endfilter shall 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 valid to execute a ret or leave instruction within a `filter` block. It is not valid 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.

Correctness:

Correct CIL guarantees the control transfer restrictions specified above.

Verifiability:

The stack shall contain exactly one item (of type `int32`).

III.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:

... → ...

Description:

Return from the `finally` or `fault` clause of an exception block (see the Exception Handling subclause 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 can 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 valid 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.

endfinally empties the evaluation stack as a side-effect.

Exceptions:

None.

Correctness:

Correct CIL guarantees the control transfer restrictions specified above.

Verifiability:

There are no additional verification requirements.

III.3.36 initblk – initialize a block of memory to a value

Format	Assembly Format	Description
FE 18	initblk	Set all bytes in a block of memory to a given byte value.

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. *end rationale*]

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

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

Correctness:

Correct CIL code ensures the restrictions specified above.

Verifiability:

The initblk instruction is never verifiable.

III.3.37 jmp – jump to method

Format	Assembly Format	Description
27 <*T*>	jmp *method*	Exit current method and jump to the 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 shall be empty when this instruction is executed. The calling convention, number and type of arguments at the destination address shall 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.

Correctness:

Correct CIL code obeys the control flow restrictions specified above.

Verifiability:

The jmp instruction is never verifiable.

III.3.38 ldarg.<length> – load argument onto the stack

Format	Assembly Format	Description
FE 09 <unsigned int16>	ldarg *num*	Load argument numbered *num* onto the stack.
0E <unsigned int8>	ldarg.s *num*	Load argument numbered *num* onto the stack, short form.
02	ldarg.0	Load argument 0 onto the stack.
03	ldarg.1	Load argument 1 onto the stack.
04	ldarg.2	Load argument 2 onto the stack.
05	ldarg.3	Load argument 3 onto the stack.

Stack Transition:

… → …, value

Description:

The ldarg *num* instruction pushes onto the evaluation stack, the *num*'th incoming argument, where arguments are numbered 0 onwards (see Partition I). The type of the value on the stack is tracked by verification as the *intermediate type* (§I.8.7) of the argument type, as specified by the current method's signature.

The ldarg.0, ldarg.1, ldarg.2, and ldarg.3 instructions are efficient encodings for loading any one of the first 4 arguments. The ldarg.s instruction is an efficient encoding for loading argument numbers 4–255.

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.)

If required, arguments are converted to the representation of their *intermediate type* (§I.8.7) when loaded onto the stack (§III.1.1.1).

[*Note:* that is arguments that hold an integer value smaller than 4 bytes, a boolean, or a character are converted to 4 bytes by sign or zero-extension as appropriate. Floating-point values are converted to their native size (type F). *end note*]

Exceptions:

None.

Correctness:

Correct CIL guarantees that *num* is a valid argument index.

Verifiability:

Verification (§III.1.8) tracks the type of the value loaded onto the stack as the *intermediate type* (§I.8.7) of the method's declared argument type.

III.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 0–255. The result is a managed pointer (type &).

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 byref parameter passing (see Partition I). In other cases, ldarg and starg should be used. *end rationale*]

Exceptions:

None.

Correctness:

Correct CIL ensures that *argNum* is a valid argument index.

Verifiability:

Verification (§III.1.8) tracks the type of the value loaded onto the stack as a managed pointer to the *verification type* (§I.8.7) of the method's declared argument type.

III.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* or some constant 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

4. For constants that shall be expressed in more than 32 bits, use the ldc.i8 instruction.
5. For constants that require 9–32 bits, use the ldc.i4 instruction followed by a conv.i8.
6. For constants that can be expressed in 8 or fewer bits, use a short form instruction followed by a conv.i8.

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.

III.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 a method pointer (§II.14.5) to the native code implementing the method described by *method* (a metadata token, either a `methoddef` or `methodref` (see Partition II)), or to some other implementation-specific description of *method* (see Note) 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). It may also be used to construct a delegate, stored in a variable, etc.

The CLI resolves the method pointer according to the rules specified in §I.12.4.1.3 (Computed destinations), except that the destination is computed with respect to the class specified by *method*.

The value returned points to native code (see Note) 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 can 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).

[*Note*: There are many options for implementing this instruction. Conceptually, this instruction places on the virtual machine's evaluation stack a representation of the address of the method specified. In terms of native code this can be an address (as specified), a data structure that contains the address, or any value that can be used to compute the address, depending on the architecture of the underlying machine, the native calling conventions, and the implementation technology of the VES (JIT, interpreter, threaded code, etc.). *end note*]

Exceptions:

`System.MethodAccessException` can be thrown when there is an invalid attempt to access a non-public method.

Correctness:

Correct CIL requires that method is a valid `methoddef` or `methodref` token.

Verifiability:

Verification tracks the method signature (§I.8.6.1.5) of the value \, which includes the number and types of parameters, the type of the *this* pointer (for an instance method), and the return type and the calling convention. [*Note*: the type of *this* pointer for an instance method is determined as described in §I.8.6.1.5 based on the resolved method definition. *end note*]

See also the `newobj` instruction.

III.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.<type> 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. The ldind.ref instruction is a shortcut for a ldobj instruction that specifies the type pointed at by *addr*, all of the other ldind instructions are shortcuts for a ldobj instruction that specifies the corresponding built-in value class.

If required, values are converted to the representation of the *intermediate type* (§I.8.7) of the <type> in the instruction when loaded onto the stack (§III.1.1.1).

[*Note:* that is integer values smaller than 4 bytes, a boolean, or a character converted to 4 bytes by sign or zero-extension as appropriate. Floating-point values are converted to `F` type. *end note*]

Correct CIL ensures that the ldind instructions are used in a manner consistent with the type of the pointer.

The address specified by *addr* shall be to a location with the natural alignment of <type> or a `NullReferenceException` might occur (but see the unaligned. prefix instruction). (Alignment is discussed in Partition I.) The results of all CIL instructions that return addresses (e.g., ldloca and ldarga) are safely aligned. For data types larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering might not run on all platforms.

The operation of the ldind instructions can 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. *end rationale*]

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

Correctness:

Correct CIL only uses an ldind instruction in a manner consistent with the type of the pointer. For ldind.ref the type pointer at by *addr* cannot be a generic parameter.

[*Note*: A ldobj instruction can be used with generic parameter types. *end note*]

Verifiability:

For `ldind.ref` *addr* shall be a managed pointer, `T&`, `T` shall be a reference type, and verification tracks the type of the result *value* as the *verification type* of `T`.

For the other instruction variants, *addr* shall be a managed pointer, `T&`, and `T` shall be *assignable-to* (§I.8.7.3) the <type> in the instruction. Verification tracks the type of the result *value* as the *intermediate type* of <type>.

III.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:

… → …, 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 localsinit 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 4 local variables. The ldloc.s instruction provides an efficient encoding for accessing local variables 4–255.

The type of the value on the stack is tracked by verification as the *intermediate type* (§I.8.7) of the local variable type, which is specified in the method header. See Partition I.

If required, local variables are converted to the representation of their *intermediate type* (§I.8.7) when loaded onto the stack (§III.1.1.1)

[*Note:* that is local variables smaller than 4 bytes, a boolean or a character are converted to 4 bytes by sign or zero-extension as appropriate. Floating-point values are converted to their native size (type `F`). *end note*]

Exceptions:

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

Correctness:

Correct CIL ensures that *indx* is a valid local index.

For the *ldloc indx* instruction, i*ndx* shall lie in the range 0–65534 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 valid, it would require a wider integer to track the number of locals in such a method. *end rationale*]

Verifiability:

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

Verification (§III.1.8) (tracks the type of the value loaded onto the stack as the *intermediate type* (§I.8.7) of the local variable.

III.3.44 ldloca.<length> – load local variable address

Format	Assembly Format	Description
FE 0D <unsigned int16>	ldloca *indx*	Load address of local variable with index *indx*.
12 <unsigned int8>	ldloca.s *indx*	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 *indx* 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 0–255. (Local variables that are the subject of ldloca shall be aligned as described in the ldind instruction, since the address obtained by ldloca can be used as an argument to ldind.)

Exceptions:

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

Correctness:

Correct CIL ensures that *indx* is a valid local index.

For the ldloca *indx* instruction, i*ndx* shall lie in the range 0–65534 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 valid, it would require a wider integer to track the number of locals in such a method. *end rationale*]

Verifiability:

Verification (§III.1.8) tracks the type of the value loaded onto the stack as a managed pointer to the *verification type* (§I.8.7) of the local variable.For verifiable code, this instruction shall guarantee that it is not loading the address of an uninitialized value – whether that initialization is done explicitly by having set the *localsinit* bit for the method, or by previous instructions (where the CLI performs definite-assignment analysis)

III.3.45 ldnull – load a null pointer

Format	Assembly Format	Description
14	ldnull	Push a 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 an ldc.i instruction, which does not exist. However, even if CIL were to include an ldc.i instruction it would still benefit verification algorithms to retain the ldnull instruction because it makes type tracking easier. *end rationale*]

Exceptions:

None.

Correctness:

Verifiability:

The ldnull instruction is always verifiable, and produces a value of the null type (§III.1.8.1.2) that is *assignable-to* (§I.8.7.3)any other reference type.

III.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 the former 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 valid to use a leave instruction to exit a `finally` block. To ease code generation for exception handlers it is valid from within a `catch` block to use a leave instruction to transfer control to any instruction within the associated `try` block.

The leave instruction can be used to exit multiple nested blocks (see Partition I).

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

Exceptions:

None.

Correctness:

Correct CIL requires the computed destination lie within the current method.

Verifiability:

See §III.1.8 for details.

III.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 → address

Description:

The localloc instruction allocates *size* (type `native unsigned int` or `U4`) bytes from the local dynamic memory pool and returns the address (an unmanaged pointer, type native int) of the first allocated byte. If the localsinit flag on the method is true, the block of memory returned is initialized to 0; otherwise, the initial value of that block of memory is unspecified. 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 shall be computed at runtime. It can be used for C's intrinsic alloca method. *end rationale*]

Exceptions:

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

Correctness:

Correct CIL requires that the evaluation stack be empty, apart from the *size* item

Verifiability:

This instruction is never verifiable.

III.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.

Correctness and Verifiability:

See Table 2: Binary Numeric Operations.

III.3.49 mul.ovf.<type> – multiply integer values with overflow check

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

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

Correctness and Verifiability:

See Table 8: Conversion Operations.

III.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.

Correctness and Verifiability:

See Table 3: Unary Numeric Operations.

III.3.51 nop – no operation

Format	Assembly Format	Description
00	Nop	Do nothing.

Stack Transition:

…, → …,

Description:

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

Exceptions:

None.

Correctness:

Verifiability:

The nop instruction is always verifiable.

III.3.52 not – bitwise complement

Format	Assembly Format	Description
66	Not	Bitwise complement.

Stack Transition:

…, value → …, result

Description:

The not instruction computes the bitwise complement of the integer value on top of the stack and leaves 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 are encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table 5: Integer Operations.

III.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 are encapsulated in Table 5: Integer Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table 5: Integer Operations.

III.3.54 pop – remove the top element of the stack

Format	Assembly Format	Description
26	pop	Pop *value* from the stack.

Stack Transition:

…, value → …

Description:

The pop instruction removes the top element from the stack.

Exceptions:

None.

Correctness:

Verifiability:

No additional requirements.

III.3.55 rem – compute remainder

Format	Assembly Format	Description
5D	rem	Remainder when dividing one value by another.

Stack Transition:

…, value1, value2 → …, result

Description:

The rem instruction divides *value1* by *value2* and pushes the remainder *result* on the stack.

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

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

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

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

For floating-point operands

rem is defined similarly as for integer operands, except that, if *value2* is zero or *value1* is infinity, *result* is NaN. If *value2* is `infinity`, *result* is *value1*. 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 `System.DivideByZeroException` if *value2* is zero.

Integral operations can throw `System.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)
```

Correctness and Verifiability:

See Table 2: Binary Numeric Operations.

III.3.56 rem.un – compute integer remainder, unsigned

Format	Assembly Format	Description
5E	rem.un	Remainder when dividing one unsigned value by another.

Stack Transition:

…, value1, value2 → …, result

Description:

The rem.un instruction divides *value1* by *value2* and pushes the remainder *result* on the stack. (rem.un treats its arguments as unsigned integers, while rem treats them as signed integers.)

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. 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 `System.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)
```

Correctness and Verifiability:

See Table 5: Integer Operations.

III.3.57 ret – return from method

Format	Assembly Format	Description
2A	Ret	Return from method, possibly with 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 shall 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.

Correctness:

Correct CIL obeys the control constraints describe above.

Verifiability:

Verification requires that the type of *retVal* is *verifier-assignable-to* the declared return type of the current method. [*Note*: as the operation is stack-to-stack no representation changes occur. *end note*]

III.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` or `native int`. The return value is unspecified if *shiftAmount* is greater than or equal to the width of *value*. See Table III.6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Correctness and Verifiability:

See Table 5: Integer Operations.

III.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` 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 *result*. See Table III.6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Correctness and Verifiability:

See Table 5: Integer Operations.

III.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 III.6: Shift Operations for details of which operand types are allowed, and their corresponding result type.

Exceptions:

None.

Correctness and Verifiability:

See Table 5: Integer Operations.

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

Format	Assembly Format	Description
FE 0B <unsigned int16>	starg *num*	Store *value* to the argument numbered *num*.
10 <unsigned int8>	starg.s *num*	Store *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 shall 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 a value smaller than 4 bytes whose *intermediate type* is `int32` 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. (See §III.1.1.1, *Numeric data types*.)

Exceptions:

None.

Correctness:

Correct CIL requires that *num* is a valid argument slot. In addition to the stores allowed by Verified CIL, Correct CIL also allows a `native int` to be stored as a byref (`&`); in which case following the store the value will be tracked by garbage collection.

Verifiability:

Verification checks that the type of *value* is *verifier-assignable-to* (§III.1.8.1.2.3) the type of the argument, as specified in the current method's signature.

III.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 value *val* at address *addr* (an unmanaged pointer, type `native int`, or managed pointer, type `&`). The address specified by *addr* shall be aligned to the natural size of *val* or a `NullReferenceException` can 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 data types larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering might not run on all platforms.

Storing into locations smaller than 4 bytes truncates the value as it moves from the stack to memory. Floating-point values are rounded from their native size (type `F`) to the size associated with the instruction. (See §III.1.1.1, *Numeric data types*.)

The stind.ref instruction is a shortcut for a stobj instruction that specifies the type pointed at by *addr*, all of the other stind instructions are shortcuts for a stobj instruction that specifies the corresponding built-in value class.

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 can be altered by an immediately preceding volatile. or unaligned. prefix instruction.

Exceptions:

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

Correctness:

Correct CIL ensures that *addr is* a pointer to `T` and the type of *val* is *verifier-assignable-to* `T`. For stind.ref the type pointer at by *addr* cannot be a generic parameter. [*Note*: A stobj instruction can be used with generic parameter types. *end note*]

Verifiability:

For verifiable code, *addr* shall be a managed pointer, `T&`, and the type of *val* shall be *verifier-assignable-to* `T` .

III.3.63 stloc – pop value from stack to local variable

Format	Assembly Format	Description
FE 0E <unsigned int16>	stloc *indx*	Pop a value from stack into local variable *indx*.
13 <unsigned int8>	stloc.s *indx*	Pop a value from stack into local variable *indx*, short form.
0A	stloc.0	Pop a value from stack into local variable 0.
0B	stloc.1	Pop a value from stack into local variable 1.
0C	stloc.2	Pop a value from stack into local variable 2.
0D	stloc.3	Pop a value from stack into local variable 3.

Stack Transition:

…, value → …

Description:

The stloc *indx* instruction pops the top value off the evaluation stack and moves it into local variable number *indx* (see Partition I), where local variables are numbered 0 onwards. The type of *value* shall 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 4 local variables; the stloc.s instruction provides an efficient encoding for local variables 4–255.

Storing into locals that hold a 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. (See §III.1.1.1, *Numeric data types*.)

Exceptions:

None.

Correctness:

Correct CIL requires that *indx* be a valid local index. For the stloc *indx* instruction, i*ndx* shall lie in the range 0–65534 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 valid, it would require a wider integer to track the number of locals in such a method. *end rationale*]

Verifiability:

Verification also checks that the type of *value* is *verifier-assignable-to* the type of the local, as specified in the current method's locals signature.

III.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 are encapsulated in Table III.2: Binary Numeric Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table2: Binary Numeric Operations.

III.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 shall fit in same size.
DB	sub.ovf.un	Subtract native unsigned int from a native unsigned int. Unsigned result shall 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 are 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:

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

Correctness and Verifiability:

See Table 7: Overflow Arithmetic Operations.

III.3.66 switch – table switch based 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, and so on). 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 shall use the leave instruction instead; see Partition I for details).

Exceptions:

None.

Correctness:

Correct CIL obeys the control transfer constraints listed above.

Verifiability:

Verification requires the type-consistency of the stack, locals and arguments for every possible way of reaching all destination instructions. See §III.1.8 for more details.

III.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 III.5: Integer Operations.

Exceptions:

None.

Correctness and Verifiability:

See Table III.5: Integer Operations.

III.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 CTS. 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. *end rationale*]

III.4.1 box – convert a boxable value to its boxed form

Format	Assembly Format	Description
8C <*T*>	box *typeTok*	Convert a boxable value to its boxed form

Stack Transition:

…, `val` → …, `obj`

Description:

If *typeTok* is a value type, the box instruction converts *val* to its boxed form. When *typeTok* is a non-nullable type (§I.8.2.4), this is done by creating a new object and copying the data from *val* into the newly allocated object. If it is a nullable type, this is done by inspecting *val*'s HasValue property; if it is false, a null reference is pushed onto the stack; otherwise, the result of boxing *val*'s Value property is pushed onto the stack.

If *typeTok* is a reference type, the box instruction does returns *val* unchanged as *obj*.

If *typeTok* is a generic parameter, the behavior of box instruction depends on the actual type at runtime. If this type is a value type it is boxed as above, if it is a reference type then *val* is not changed. However the type tracked by verification is always "boxed" *typeTok* for generic parameters, regardless of whether the actual type at runtime is a value or reference type.

typeTok is a metadata token (a `typedef`, `typeref`, or `typespec`) indicating the type of *val*. *typeTok* can represent a value type, a reference type, or a generic parameter.

Exceptions:

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

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

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token. The type operand *typeTok* shall represent a boxable type (§I.8.2.4).

Verifiability:

The top-of-stack shall be *verifier-assignable-to* the type represented by *typeTok*. When *typeTok* represents a non-nullable value type or a generic parameter, the resulting type is "boxed" *typeTok*; when *typeTok* is Nullable<T>, the resulting type is "boxed" *T*. When *typeTok* is a

reference type, the resulting type is *typeTok*. The type operand *typeTok* shall not be a byref-like type.

III.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 an object.

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 exact 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 can 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`, `methodref` or `methodspec` 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. The CLI resolves the method to be called according to the rules specified in §I.12.4.1.3 (Computed destinations).

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, and so on.

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) shall be pushed first. The signature carried in the metadata does not contain an entry in the parameter list for the `this` pointer, but the calling convention always indicates whether one is required and if its signature is explicit or inferred (see §I.8.6.1.5 and §II.15.3) [*Note*: For calls to methods on value types, the `this` pointer may be a managed pointer, not an instance reference (§I.8.6.1.5). *end note*]

The arguments are passed as though by implicit `starg` (§III.3.61) instructions, see *Implicit argument coercion* §III.1.6.

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

Exceptions:

`System.MethodAccessException` can be thrown when there is an invalid attempt to access a non-public method.

`System.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 base classes. This is typically detected when CIL is converted to native code, rather than at runtime.

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

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

Correctness:

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 addition to the arguments types allowed by Verified CIL, Correct CIL also allows a `native int` to be passed as a byref (`&`); in which case following the store the value will be tracked by garbage collection.

Verifiability:

In its typical use, callvirt is verifiable if:

(a) the above restrictions are met;

(b) the verification type of *obj* is *verifier-assignable-to* (§III.1.8.1.2.3) with the `this` signature of the method's signature;

(c) the types of the arguments on the stack are *verifier-assignable-to* (§III.1.8.1.2.3) the parameter signatures of the method's signature; and

(d) the method is accessible from the call site.

If *returnVal* is present its type is tracked as the *intermediate type* of the called method's signature return type.

A callvirt annotated by tail. has additional considerations – see §III.1.8.

III.4.3 castclass – cast an object to a class

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

Stack Transition:

…, obj → …, obj2

Description:

typeTok is a metadata token (a `typeref`, `typedef` or `typespec`), indicating the desired class. If *typeTok* is a non-nullable value type or a generic parameter type it is interpreted as "boxed" *typeTok*. If *typeTok* is a nullable type, `Nullable<T>`, it is interpreted as "boxed" `T`.

The castclass instruction determines if *obj* (of type `O`) is an instance of the type *typeTok,* termed "casting".

If the actual type (not the verifier tracked type) of *obj* is *verifier-assignable-to* the type *typeTok* the cast succeeds and *obj* (as *obj2*) is returned unchanged while verification tracks its type as *typeTok*.

Unlike coercions (§III.1.6) and conversions (§III.3.27), a cast never changes the actual type of an object and preserves object identity (see Partition I).

If the cast fails then an `InvalidCastException` is thrown.

If *obj* is null, castclass succeeds and returns null. This behavior differs semantically from isinst where if *obj* is null, isinst *fails* and returns null.

Exceptions:

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

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

Correctness:

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

Verifiability:

Verification tracks the type of *obj2* as *typeTok*.

III.4.4 cpobj – copy a value from one address to another

Format	Assembly Format	Description
70 <*T*>	cpobj *typeTok*	Copy a value type from *src* to *dest*.

Stack Transition:

…, *dest*, *src* → …,

Description:

The cpobj instruction copies the value at the address specified by *src* (an unmanaged pointer, `native int`, or a managed pointer, `&`) to the address specified by *dest* (also a pointer). *typeTok* can be a `typedef`, `typeref`, or `typespec`. **The behavior is unspecified if the type of the location referenced by** *src* **is not** *assignable-to* (§I.8.7.3) **the type of the location referenced by** *dest*.

If *typeTok* is a reference type, the cpobj instruction has the same effect as ldind.ref followed by stind.ref.

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

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

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

Verifiability:

The tracked types of the destination (*dest*) and source (*src*) values shall both be managed pointers (`&`) to values whose types we denote *destType* and *srcType*, respectively. Finally, *srcType* shall be *assignable-to* (§I.8.7.3) *typeTok*, and *typeTok* shall be *assignable-to* (§I.8.7.3) *destType*. In the case of an Enum, its type is that of the underlying, or base, type of the Enum.

III.4.5 initobj – initialize the value at an address

Format	Assembly Format	Description
FE 15 <*T*>	initobj *typeTok*	Initialize the value at address *dest*.

Stack Transition:

…, dest → …,

Description:

The initobj instruction initializes an address with a default value. *typeTok* is a metadata token (a `typedef`, `typeref`, or `typespec`). *dest* is an unmanaged pointer (`native int`), or a managed pointer (`&`). If *typeTok* is a value type, the initobj instruction initializes each field of *dest* to null or a zero of the appropriate built-in type. If *typeTok* is a value type, then after this instruction is executed, the instance is ready for a constructor method to be called. If *typeTok* is a reference type, the initobj instruction has the same effect as ldnull followed by stind.ref.

Unlike newobj, the initobj instruction does not call any constructor method.

Exceptions:

None.

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

Verifiability:

The type of the destination value on top of the stack shall be a managed pointer to some type *destType*, and *typeTok* shall be *assignable-to destType*. If *typeTok* is a non-reference type, the definition of subtyping implies that *destType* and *typeTok* shall be equal .

III.4.6 isinst – test if an object is an instance of a class or interface

Format	Assembly Format	Description
75 <*T*>	isinst *typeTok*	Test if *obj* is an instance of *typeTok, returning null or an instance of that class or interface.*

Stack Transition:

…, obj → …, result

Description:

typeTok is a metadata token (a `typeref`, `typedef` or `typespec`), indicating the desired class. If *typeTok* is a non-nullable value type or a generic parameter type it is interpreted as "boxed" *typeTok*. If *typeTok* is a nullable type, `Nullable<T>`, it is interpreted as "boxed" `T`.

The isinst instruction tests whether *obj* (type `O`) is an instance of the type *typeTok*.

If the actual type (not the verifier tracked type) of *obj* is *verifier-assignable-to* the type *typeTok* then isinst *succeeds* and *obj* (as *result*) is returned unchanged while verification tracks its type as *typeTok*. Unlike coercions (§III.1.6) and conversions (§III.3.27), isinst never changes the actual type of an object and preserves object identity (see Partition I).

If *obj* is null, or *obj* is not *verifier-assignable-to* the type *typeTok*, isinst *fails* and returns null.

Exceptions:

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

Correctness:

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

Verifiability:

Verification tracks the type of *result* as *typeTok*.

III.4.7 ldelem – load element from array

Format	Assembly Format	Description
A3 <*T*>	ldelem *typeTok*	Load the element at *index* onto the top of the stack.

Stack Transition:

…, array, index → …, value

Description:

The ldelem instruction loads the value of the element with index *index* (of type `native int` or `int32`) in the zero-based one-dimensional array *array*, and places it on the top of the stack. The type of the return value is indicated by the type token *typeTok* in the instruction.

If required elements are converted to the representation of their *intermediate type* (§I.8.7) when loaded onto the stack (§III.1.1.1).

[*Note*: that is elements that are smaller than 4 bytes, a boolean or a character are converted to 4 bytes by sign or zero-extension as appropriate. Floating-point values are converted to their native size (type `F`). *end note*]

Exceptions:

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

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

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

array shall be either null or a single dimensional, zero-based array.

Verifiability:

Verification requires that:

- the tracked type of *array* is `T[]`, for some `T`;
- `T` is *array-element-compatible-with* (§I.8.7.1) *typeTok*; and
- the type of *index* is `int32` or `native int`.

Verification tracks the type of the result *value* as *typeTok*.

III.4.8 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 a native int.
9A	ldelem.ref	Load the element at *index* onto the top of the stack as an O. The type of the O is the same as the element type of the array pushed on the CIL stack.

Stack Transition:

…, array, index → …, value

Description:

The ldelem.<type> 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. For `ldelem.ref` the type of the return *value* is the element type of *array*, for the other instruction variants it is the <type> indicated by the instruction.

All variants are equivalent to the `ldelem` instruction (§III.4.7) with an appropriate *typeTok*.

[*Note*: For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a `Get` method. *end note*]

If required elements are converted to the representation of their *intermediate type* (§I.8.7) when loaded onto the stack (§III.1.1.1).

[*Note*: that is elements that are smaller than 4 bytes, a boolean or a character are converted to 4 bytes by sign or zero-extension as appropriate. Floating-point values are converted to their native size (type `F`). *end note*]

Exceptions:

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

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

Correctness:

Correct CIL code requires that *array* is either null or a zero-based, one-dimensional array whose declared element type is *array-element-compatible-with* (§I.8.7.1) the type for this particular instruction suffix.

Verifiability:

Verification requires that:

- the tracked type of *array* is `T[]`, for some `T`;
- for `ldelem.ref T` is a reference type, for other instruction variants `T` is *array-element-compatible-with* the type in the instruction; and
- the type of *index* is `int32` or `native int`.

Verification tracks the type of the result *value* as `T` for `ldelem.ref`, or as the <type> in the instruction for the other variants.

III.4.9 ldelema – load address of an element of an array

Format	Assembly Format	Description
8F *<T>*	ldelema *typeTok*	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 *verifier-assignable-to typeTok)* 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 &).

[*Note*: For one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides an Address method. *end note*]

If this instruction is prefixed by the readonly. prefix, it produces a controlled-mutability managed pointer (§III.1.8.1.2.2).

Exceptions:

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

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

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

Correctness:

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

Verifiability:

Verification requires that:

- the tracked type of *array* is T[], for some T, or the Null type (§III.1.8.1.2);
- a managed pointer to T is *pointer-element-compatible-with* (§I.8.7.1) a managed pointer to *typeTok*; and
- the type of *index* is int32 or native int.

Verification tracks the type of the result *address* as a managed pointer to the *verification type* of *typeTok*.

III.4.10 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 → …, value

Description:

The ldfld instruction pushes onto the stack the value of a field of *obj*. *obj* shall 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 shall 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 can be either an instance field (in which case *obj* shall not be null) or a static field.

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

If required field values are converted to the representation of their *intermediate type* (§I.8.7) when loaded onto the stack (§III.1.1.1).

[*Note*: That is field values that are smaller than 4 bytes, a boolean or a character are converted to 4 bytes by sign or zero-extension as appropriate. Floating-point values are converted to their native size (type `F`). *end note*]

Exceptions:

`System.FieldAccessException` is thrown if *field* is not accessible.

`System.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.

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

Correctness:

Correct CIL ensures that *field* is a valid token referring to a field, and that the type of *obj* is *compatible-with* the *Class* of *field*.

Verifiability:

For verifiable code, *obj* shall not be an unmanaged pointer.

The tracked type of *obj* shall have, or be a managed pointer to a type which has, a static or instance *field*.

It is not verifiable to access an overlapped object reference field.

A field is accessible only if every field that overlaps it is also accessible.

Verification tracks the type of the *value* on the stack as the *intermediate type* (§I.8.7) of the *field* type.

III.4.11 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 shall refer to a field member. The field can be either an instance field (in which case *obj* shall not be null) or a static field.

Exceptions:

`System.FieldAccessException` is thrown if *field* is not accessible.

`System.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.

`System.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.

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

Correctness:

Correct CIL ensures that *field* is a valid `fieldref` token and that the type of *obj* is *compatible-with* the *Class* of *field*.

Verifiability:

For verifiable code, *obj* shall not be an unmanaged pointer.

The tracked type of *obj* shall have, or be a managed pointer to a type which has, a static or instance *field*.

For verifiable code, *field* cannot be init-only.

It is not verifiable to access an overlapped object reference field.

A field is accessible only if every field that overlaps it is also accessible.

Verification (§III.1.8) tracks the type of the value loaded onto the stack as a managed pointer to the *verification type* (§I.8.7) of *field*.

Remark:

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 might lead to unpredictable behavior.

III.4.12 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:

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

Correctness:

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

III.4.13 ldobj – copy a value from an address to the stack

Format	Assembly Format	Description
71 <*T*>	ldobj *typeTok*	Copy the value stored at address *src* to the stack.

Stack Transition:

…, src → …, val

Description:

The `ldobj` instruction copies a value to the evaluation stack. *typeTok* is a metadata token (a `typedef`, `typeref`, or `typespec`). *src* is an unmanaged pointer (`native int`), or a managed pointer (`&`). If *typeTok* is not a generic parameter and either a reference type or a built-in value class, then the ldind instruction provides a shorthand for the ldobj instruction..

[*Rationale*: The ldobj instruction can be used to pass a value type as an argument. *end rationale*]

If required values are converted to the representation of the *intermediate type* (§I.8.7) of *typeTok* when loaded onto the stack (§III.1.1.1).

[*Note*: That is integer values of less than 4 bytes, a boolean or a character are converted to 4 bytes by sign or zero-extension as appropriate. Floating-point values are converted to `F` type. *end note*]

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

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

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

Correctness:

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

[*Note*: Unlike the ldind instruction a ldobj instruction can be used with a generic parameter type. *end note*]

Verifiability:

The tracked type of the source value on top of the stack shall be a managed pointer to some type *srcType*, and *srcType* shall be a *assignable-to* the type *typeTok*. Verification tracks the type of the result *val* as the *intermediate type* of *typeTok*.

III.4.14 ldsfld – load static field of a class

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

Stack Transition:

…, → …, value

Description:

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

The ldsfld instruction can have a volatile. prefix.

If required field values are converted to the representation of their *intermediate type* (§I.8.7) when loaded onto the stack (§III.1.1.1).

[*Note*: That is field values that are smaller than 4 bytes, a boolean or a character are converted to 4 bytes by sign or zero-extension as appropriate. Floating-point values are converted to their native size (type `F`). *end note*]

Exceptions:

`System.FieldAccessException` is thrown if *field* is not accessible.

`System.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.

Correctness:

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

Verifiability:

Verification tracks the type of the *value* on the stack as the *intermediate type* (§I.8.7) of the *field* type.

III.4.15 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* can 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:

`System.FieldAccessException` is thrown if *field* is not accessible.

`System.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.

Correctness:

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.

Verifiability:

For verifiable code, *field* cannot be init-only.

If *field* refers to a type whose memory is managed, verification (§III.1.8) tracks the type of the value loaded onto the stack as a managed pointer to the *verification type* (§I.8.7) of *field*. If *field* refers to a type whose memory is unmanaged, verification (§III.1.8) tracks the type of the value loaded onto the stack as an unmanaged pointer.

Remark:

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 can lead to unpredictable behavior.

III.4.16 ldstr – load a literal string

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

Stack Transition:

…, → …, string

Description:

The ldstr instruction pushes a new string object representing the literal stored in the metadata as *string* (which is a string literal).

By default, 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"). This behavior can be controlled using the `System.Runtime.CompilerServices. CompilationRelaxationsAttribute` and the `System.Runtime.CompilerServices. CompilationRelaxations.NoStringInterning` (see Partition IV).

Exceptions:

None.

Correctness:

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

Verifiability:

There are no additional verification requirements.

III.4.17 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 shall be one of:

A `methoddef`, `methodref` or `methodspec`: pushes a `RuntimeMethodHandle`

A `typedef`, `typeref`, or `typespec` : 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.

Correctness:

Correct CIL requires that *token* describes a valid metadata token of the kinds listed above

Verifiability:

There are no additional verification requirements.

III.4.18 ldvirtftn – load a virtual method pointer

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

Stack Transition:

… object → …, ftn

Description:

The ldvirtftn instruction pushes a method pointer (§II.14.5) to the native code implementing the virtual method associated with *object* and described by the method reference *method* (a metadata token, a `methoddef`, `methodref` or `methodspec`; see Partition II), or to some other implementation-specific description of the *method* associated with object (see Note), 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). It may also be used to construct a delegate, stored in a variable, etc.

The value returned points to native code (see Note) using the calling convention specified by *method*. Thus a method pointer can be passed to unmanaged native code (e.g., as a callback routine) if that routine expects the corresponding calling convention. [*Note*: that the address computed by this instruction can 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). *end note*]

[*Note*: There are many options for implementing this instruction. Conceptually, this instruction places on the virtual machine's evaluation stack a representation of the address of the method specified. In terms of native code this can be an address (as specified), a data structure that contains the address, or any value that can be used to compute the address, depending on the architecture of the underlying machine, the native calling conventions, and the implementation technology of the VES (JIT, interpreter, threaded code, etc.). *end note*]

Exceptions:

`System.MethodAccessException` can be thrown when there is an invalid attempt to access a non-public method.

`System.NullReferenceException` is thrown if *object* is null.

Correctness:

Correct CIL ensures that *method* is a valid `methoddef`, `methodref` or `methodspec` token. Also that *method* references a non-static method that is defined for *object*.

Verifiability:

Verification requires that tracked type of *object* combined with *method* identify a final virtual method. [*Rationale*: If the identified method is not final then the exact type of its *this* pointer cannot be statically determined. *end rationale*]

There is a defined exception to the above requirement as described for `newobj` (§III.4.21).

Verification tracks the method signature (§I.8.6.1.5) of the value, which includes the number and types of parameters, the type of the *this* pointer, and the return type and the calling convention. [*Note*: the type of the this pointer is determined in §I.8.6.1.5 based on the resolved method definition. *end note*]

See also the `newobj` instruction.

III.4.19 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* shall be a pointer (type `&`, or `native int`) that holds the address of a piece of data. *class* is the class token (a `typeref`, `typedef` or `typespec`; 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*. This instruction enables the passing of dynamically typed references as arguments. The callee can use the refanytype and refanyval instructions to retrieve the type (*class*) and address (*ptr*) respectively of the parameter.

Exceptions:

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

Correctness:

Correct CIL ensures that *class* is a valid `typeref` or `typedef` or `typespec` token describing some type and that *ptr* is a pointer to exactly that type.

Verifiability:

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*.

III.4.20 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 *etype*, a metadata token (a `typeref`, `typedef` or `typespec`; see Partition II). *numElems* (of type `native int` or `int32`) 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:

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

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

Correctness:

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

Verifiability:

.*numElems* shall be of type `native int` or `int32`.

III.4.21 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 shall 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 *ctor* 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:

`System.InvalidOperationException` is thrown if *ctor*'s class is abstract.

`System.MethodAccessException` is thrown if *ctor* is inaccessible.

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

`System.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.

Correctness:

Correct CIL ensures that *ctor* is a valid `methodref` or `methoddef` token, and that the arguments on the stack are *assignable-to* (§I.8.7.3) the parameters of the constructor.

Verifiability:

Verification depends on whether a delegate or other object is being created. There are three cases, in order:

1. If the newobj instruction is part of a dup; ldvirtftn; newobj instruction sequence and the *ctor* metadata token references a delegate type then a delegate for a virtual function is being created;
2. If the newobj instruction is part of a ldftn; newobj instruction sequence and the *ctor* metadata token references a delegate type then a delegate for a static or non-virtual instance function is being created;
3. Otherwise if the *ctor* metadata token does not references a delegate type then some other object is being created.

No other cases are verifiable. The different verification rules for the three cases follow.

Verifiability of virtual dispatch delegate creation:

When a newobj instruction is part of a:

dup

ldvirtftn *function*

newobj *ctor*

instruction sequence then verification checks that:

1. there is a *target* on the stack prior to the dup instruction of type T;
2. *function* is a `methoddef`, `methodref` or `methodspec` metadata token for a virtual method on type T;
3. *ctor* is a `methoddef` or `methodref` metadata token marked as a constructor for a delegate type *deltype*;
4. *ctor* is accessible from the newobj site;
5. the signature of *function* is *delegate-assignable-to* the signature of *deltype* (i.e. the signature of the `Invoke` method of *deltype*);
6. the *verification type* of *target* is *verifier-assignable-to* (§III.1.8.1.2.3) the *this* signature of *function*; and
7. no branch instructions target the ldvirtftn or newobj instructions within the sequence.

Verification tracks the type of *obj* as *deltype*.

Verifiability of interface dispatch delegate creation for static and instance methods:

When a newobj instruction is part of a:

ldftn *function*

newobj *ctor*

instruction sequence then verification checks that:

1. *function* is a `methoddef`, `methodref` or `methodspec` metadata token for a static or non-virtual instance method;
2. there is a *target* on the stack prior to the ldftn instruction and the *verification type* of *target* is either:
   a. *verifier-assignable-to* (§III.1.8.1.2.3) the *this* signature of *function*, if *function* refers to an instance method, or
   b. null (i.e. the result of ldnull), if *function* refers to a static method
3. *ctor* is a `methoddef` or `methodref` metadata token marked as a constructor for a delegate type *deltype*;
4. *ctor* is accessible from the newobj site;
5. the signature of *function* is *delegate-assignable-to* the signature of *deltype* (i.e. the signature of the `Invoke` method of *deltype*); and
6. when *function* is a non-final virtual method and the *target* on the stack is not a boxed valued type, verification checks that *target* is the result of ldarg.s 0, ldarg 0 or ldarg.0 and the creator's body does not contain starg.s 0, starg 0 or ldarga.s 0, ldarga 0. [*Note*: This mirrors the requirement, and rationale, for the call instruction (§III.3.19). *end note*]; and
7. no branch instructions target the newobj instruction within the sequence.

Verification tracks the type of *obj* as *deltype*.

Verifiability of creation of non-delegate objects:

Verification checks that:

1. *ctor* is a `methoddef` or `methodref` metadata token marked as a constructor for a non-delegate type T;
2. *ctor* is accessible from the newobj site; and
3. the types of the arguments; *arg1*, … *argN*; on the stack are *verifier-assignable-to* (§III.1.8.1.2.3) the parameter signatures of *ctor*'s signature.

Verification tracks the type of *obj* as T.

III.4.22 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.

Correctness:

Correct CIL ensures that *TypedRef* is a valid typed reference (created by a previous call to mkrefany).

Verifiability:

The `refanytype` instruction is always verifiable.

III.4.23 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* shall match the type specified by *type* (a metadata token, either a `typedef`, `typeref` or a `typespec`; see Partition II). See the mkrefany instruction.

Exceptions:

`System.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*)

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

Correctness:

Correct CIL ensures that *TypedRef* is a valid typed reference (created by a previous call to mkrefany).

Verifiability:

The refanyval instruction is always verifiable.

III.4.24 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. A rethrow does not change the stack trace in the object.

Exceptions:

The original exception is thrown.

Correctness:

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, an exception will be thrown, but precisely which exception, is undefined

Verifiability:

There are no additional verification requirements.

III.4.25 sizeof – load the size, in bytes,of a type

Format	Assembly Format	Description
FE 1C <*T*>	sizeof *typeTok*	Push the size, in bytes, of a type as an `unsigned int32`.

Stack Transition:

…, → …, `size (4 bytes, unsigned)`

Description:

Returns the size, in bytes, of a type. *typeTok* can be a generic parameter, a reference type or a value type.

For a reference type, the size returned is the size of a reference value of the corresponding type, not the size of the data stored in objects referred to by a reference value.

[*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 type – including any padding the implementation chooses to add. Specifically, array elements lie `sizeof` bytes apart. *end rationale*]

Exceptions:

None.

Correctness:

typeTok shall be a `typedef`, `typeref`, or `typespec` metadata token.

Verifiability:

It is always verifiable.

III.4.26 stelem – store element to array

Format	Assembly Format	Description
A4 <*T*>	stelem *typeTok*	Replace array element at *index* with the *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 native int or int32) in the one-dimensional array *array*, with *value*. Arrays are objects and hence are represented by a value of type O. The type of value must be *array-element-compatible-with typeTok* in the instruction.

Storing into arrays that hold values smaller than 4 bytes whose *intermediate type* is int32 truncates the value as it moves from the stack to the array. Floating-point values are rounded from their native size (type F) to the size associated with the array. (See §III.1.1.1, *Numeric data types*.)

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

Exceptions:

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

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

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

Correctness:

typeTok shall be a valid typedef, typeref, or typespec metadata token.

array shall be null or a single dimensional array.

Verifiability:

Verification requires that:

- the tracked type of *array* is T[], for some T;
- the tracked type of *value* is *array-element-compatible-with* (§I.8.7.1) *typeTok;*
- *typeTok* is *array-element-compatible-with* T; and
- the type of *index* is int32 or native int.

III.4.27 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 native int *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.<type> 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`.

Storing into arrays that hold values smaller than 4 bytes whose *intermediate type* is `int32` truncates the value as it moves from the stack to the array. Floating-point values are rounded from their native size (type `F`) to the size associated with the array. (See §III.1.1.1, *Numeric data types*.)

All variants, except stelem.ref, are equivalent to the `stelem` instruction (§III.4.26) with an appropriate *typeTok*.

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 can throw the `ArrayTypeMismatchException`. This behavior differs from stelem.

[*Note*: for one-dimensional arrays that aren't zero-based and for multidimensional arrays, the array class provides a `StoreElement` method. *end note*]

Exceptions:

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

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

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

Correctness:

Correct CIL requires that *array* be a zero-based, one-dimensional array, and that the type in the instruction is *array-element-compatible-with* its declared element type.

Verifiability:

Verification requires that:

- the tracked type of *array* is `T[]`, for some `T`;
- for `stelem.ref` the tracked type of *value* is a reference type and is *(array-element-)compatible-with* `T`;
- for other instruction variants the tracked type of *value* is *array-element-compatible-with* (§I.8.7.1) the type in the instruction, and the type in the instruction is *array-element-compatible-with* `T`; and
- the type of *index* is `int32` or `native int`.

III.4.28 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 *value*.

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.

Storing into fields that hold a value smaller than 4 bytes 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. (See §III.1.1.1, *Numeric data types*.)

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

Exceptions:

`System.FieldAccessException` is thrown if *field* is not accessible.

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

`System.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.

Correctness:

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, subject to implicit conversion as specified in §III.1.6.

Verifiability:

For verifiable code, *obj* shall 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 can lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by verification. *end note*]

The tracked type of *obj* shall have, or be a managed pointer to a type which has, a static or instance *field*.

It is not verifiable to access an overlapped object reference field.

A field is accessible only if every field that overlaps it is also accessible.

Verification also checks that the type of *value* is *verifier-assignable-to* the type of the field.

III.4.29 stobj – store a value at an address

Format	Assembly Format	Description
81 <*T*>	stobj *typeTok*	Store a value of type *typeTok* at an address.

Stack Transition:

…, dest, src → …,

Description:

The stobj instruction copies the value *src* to the address *dest*. If *typeTok* is not a generic parameter and either a reference type or a built-in value class, then the stind instruction provides a shorthand for the stobj instruction.

Storing values smaller than 4 bytes truncates the value as it moves from the stack to memory. Floating-point values are rounded from their native size (type `F`) to the size associated with *typeTok*. (See §III.1.1.1, *Numeric data types*.)

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

Exceptions:

`System.NullReferenceException` can be thrown if an invalid address is detected.

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

Correctness:

Correct CIL ensures that *dest is* a pointer to `T` and the type of *src* is *verifier-assignable-to* `T`.

typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

[*Note*: Unlike the stind instruction a stobj instruction can be used with a generic parameter type. *end note*]

Verifiability:

Let the tracked type of the value on top of the stack be some type *srcType*. The value shall be initialized (when *srcType* is a reference type). The tracked type of the destination address *dest* on the preceding stack slot shall be a managed pointer (of type `destType&`) to some type *destType*. Finally, *srcType* shall be *verifier-assignable-to typeTok*.

III.4.30 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 shall refer to a static field member. stsfld pops the value off the stack and updates the static field with that value.

Storing into fields that hold a value smaller than 4 bytes 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. (See §III.1.1.1, *Numeric data types*.)

The stsfld instruction can have a volatile. prefix.

Exceptions:

`System.FieldAccessException` is thrown if *field* is not accessible.

`System.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.

Correctness:

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, subject to implicit conversion as specified in §III.1.6.

Verifiability:

Verification checks that the type of *val* is *verifier-assignable-to* the type of the field.

[*Note:* Using stsfld to change the value of a static, init-only field outside the body of the class initializer can lead to unpredictable behavior. It cannot, however, compromise memory integrity or type safety so it is not tested by verification. *end note*]

III.4.31 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 and empties the stack. For details of the exception mechanism, see Partition I.

[*Note:* While the CLI permits any object to be thrown, the CLS describes a specific exception class that shall be used for language interoperability. *end note*]

Exceptions:

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

Correctness:

Correct CIL ensures that *object* is always either null or an object reference (i.e., of type `O`).

Verifiability:

There are no additional verification requirements.

III.4.32 unbox – convert boxed value type to its raw form

Format	Assembly Format	Description
79 <*T*>	unbox *valuetype*	Extract a value-type 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 controlled-mutability managed pointer (§III.1.8.1.2.2), type `&`), its unboxed form. *valuetype* is a metadata token (a `typeref`, `typedef` or `typespec`). The type of *valuetype* contained within *obj* must be *verifier-assignable-to valuetype*.

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.

[*Note*: Typically, unbox simply computes the address of the value type that is already present inside of the boxed object. This approach is not possible when unboxing nullable value types. Because Nullable<*T*> values are converted to boxed *T*s during the box operation, an implementation often must manufacture a new Nullable<*T*> on the heap and compute the address to the newly allocated object. *end note*]

Exceptions:

`System.InvalidCastException` is thrown if *obj* is not a boxed value type, *valuetype* is a Nullable<*T*> and *obj* is not a boxed *T*, or if the type of the value contained in *obj* is not *verifier-assignable-to* (§III.1.8.1.2.3) *valuetype*.

`System.NullReferenceException` is thrown if *obj* is null and *valuetype* is a non-nullable value type (Partition I.8.2.4).

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

Correctness:

Correct CIL ensures that *valueType* is a `typeref`, `typedef` or `typespec` metadata token for some boxable value type, and that *obj* is always an object reference (i.e., of type `O`). If *valuetype* is the type Nullable<*T*>, the boxed instance shall be of type *T*.

Verifiability:

Verification requires that the type of *valuetype* contained within *obj* must be *verifier-assignable-to valuetype*

III.4.33 unbox.any – convert boxed type to value

Format	Assembly Format	Description
A5 <*T*>	unbox.any *typeTok*	Extract a value-type from *obj*, its boxed representation

Stack Transition:

…, obj → …, value or obj

Description:

When applied to the boxed form of a value type, the unbox.any instruction extracts the value contained within *obj* (of type `o`). (It is equivalent to unbox followed by ldobj.) When applied to a reference type, the unbox.any instruction has the same effect as castclass *typeTok*.

If *typeTok* is a *GenericParam*, the runtime behavior is determined by the actual instantiation of that parameter.

Exceptions:

`System.InvalidCastException` is thrown if *obj* is not a boxed value type or a reference type, *typeTok* is Nullable<*T*> and *obj* is not a boxed *T*, or if the type of the value contained in *obj* is not *verifier-assignable-to* (§III.1.8.1.2.3) *typeTok*.

`System.NullReferenceException` is thrown if *obj* is null and *typeTok* is a non-nullable value type (Partition I.8.2.4).

Correctness:

obj shall be of reference type and *typeTok* shall be a boxable type.

Verifiability:

Verification tracks the type of *value or obj* as the *intermediate type* of *typeTok*.

Rationale:

There are two reasons for having both unbox.any and unbox instructions:

1. Unlike the unbox instruction, for value types, unbox.any leaves a value, not an address of a value, on the stack.
2. The type operand to unbox has a restriction: it can only represent value types and instantiations of generic value types.