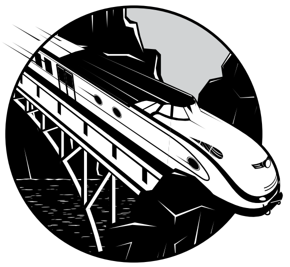
# E

# CIL ASSEMBLY FOR THE HLL PROGRAMMER



The Common Intermediate Language (CIL; previously Microsoft Intermediate Language, MSIL, or just IL) is a low-level, assembly-like language Microsoft originally developed for .NET languages. Today, languages inside and outside Microsoft use CIL for cross-platform development. Compilers for .NET languages typically produce a special CIL *bytecode* rather than true machine language. At load time, a just-in-time (JIT) compiler translates CIL bytecode into a sequence of native machine instructions that do the same work as specified by that bytecode (note that the CIL code is never interpreted). To understand how .NET compilers such as Visual C# and VB.NET/Visual Basic operate, you need a basic familiarity with the CIL bytecode instruction set. Therefore, this appendix:

- Describes the basic CIL bytecode machine architecture
- Provides an overview of CIL bytecode assembly language so that you'll be able to read the bytecode output produced by the .NET compilers

## E.1 Assembly Syntax

Unlike Java bytecode, the CIL is an actual assembly language. Various Microsoft (and other company) language compilers, such as C#, VB.NET, and F#, produce CIL assembly language that a CIL assembler converts into CIL bytecode. Microsoft calls this assembly language source code *Intermediate Language Assembly*, or *ILAsm*. This book uses the terms *CIL*, *MSIL*, and *ILAsm* interchangeably.

MSIL uses a relatively standard assembly language syntax. Each line of source code typically contains an MSIL instruction or directive, as follows:[1]

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       27 (0x1b)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           int32 V_2)
  IL_0000:  nop
  IL_0001:  ldc.i4.5
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.6
  IL_0004:  stloc.1
  IL_0005:  ldloc.0
  IL_0006:  ldloc.1
  IL_0007:  add
  IL_0008:  stloc.2
  IL_0009:  ldstr      "Hello World! k={0}"
  IL_000e:  ldloc.2
  IL_000f:  box       [mscorlib]System.Int32
  IL_0014:  call void [mscorlib]System.Console::WriteLine(string, object)
  IL_0019:  nop
  IL_001a:  ret
} // end of method program::Main
```

## E.2 The Motivation for CLR

The .NET Framework provides a runtime environment (CLR) that runs the code and provides JIT compilation, library code, and other features. The main motivation behind the CLR was to provide a common set of tools for different programming languages. Any CLR-based (.NET) language can link with modules from other CLR-based languages, and they can share data. In Microsoft's case, for example, VB.NET, C#, C++, and F# programmers can all share code and libraries written in their respective languages.

Compilers for CLR languages emit CIL instead of native machine code for portability. To write native code compilers for each of *n* languages on

1. An explanation of the MSIL instructions appears later in this appendix; this example simply demonstrates an MSIL source file.

$m$ different CPUs,[2] you'd have to write $m \times n$ different compilers. However, when all $n$ language compilers produce a portable output format (such as CIL), you only have to write $n$ compilers and $m$ translators that convert CIL to a particular CPU's native code. Writing only $m + n$ language translation products is considerably less work. This was especially important to Microsoft, as it wanted to support the aforementioned VB.NET, C#, C++, and F# languages.

Microsoft did not design the CLR for direct execution (via an interpreter). The intent was to provide a machine-independent description of a program's semantics that the operating system loader can translate into native (executable) machine code as the system needs to execute the code.

Translation to machine code can happen at three different times:

**Execution time** As the system attempts to execute the CIL bytecode, the system translates a CIL module into native code using JIT compiler techniques.

**Load time** The system translates the entire CIL program into native machine code when the program loads into memory. This is known as *ahead-of-time (AOT)* compilation.

**Prior to load time** A utility translates a file containing CIL bytecode into an executable native code file and writes the executable code to a separate file for execution (this is also an AOT compilation process).

At first blush, it might seem that AOT compilation prior to load time is always better than JIT compilation. Translating CIL into native machine code consumes CPU cycles, after all. Translating the entire program into machine code when it first loads into memory could introduce a sizeable delay before the program begins execution (especially if any optimization happens during translation). Translating the code module by module (or function by function) won't introduce as much of a delay on loading, but it will cause delays every time the program encounters a module or function that it hasn't yet translated.

If you do an AOT compilation of the entire executable and save the native code as a file (and load and execute that file without any load or runtime translation), you don't pay the JIT translation costs every time you run the application. However, it's not always the case that this approach saves considerable execution time.

There are two reasons a JIT translation could actually run faster than loading the native code. First, CIL is much more compact than native code (at least on most modern architectures), so it might be faster to load the smaller CIL file and translate the portions of the program that actually execute than to load the entire native code application from secondary storage to execute it.[3]

2. Such as the 32-bit 80x86, 64-bit x86-64, 32-bit ARM, 64-bit ARM, MIPS, and PowerPC architectures.

3. In practice, modern solid-state drives are very fast, and most large applications contain substantial data (requiring the same amount of time to load in native or CIL code), so the smaller code size may not make up for the cost of JIT translation.

The second, and main, reason CIL might run faster is that JIT translations exhibit locality of reference, whereas executable code is likely to be distributed throughout a native file (and, therefore, throughout memory once it loads into memory).

As a result, the CPU is very likely to thrash the cache and virtual memory as the program transfers control between functions. The JIT translation, on the other head, stacks up the native code (as it's generated) in sequential pages in memory, so it's more likely that the JIT-translated code will sit in active cache lines and in virtual memory pages that remain in memory. This reduces thrashing and can dramatically improve the application's performance. For this reason, most systems leave larger applications in CIL form.

## E.3 The CLR Virtual Machine Architecture

The CLR virtual machine (VM)[4] uses a typical *p-machine* (pseudo-machine) stack architecture, similar to the UCSD Pascal p-machine from the 1970s and 1980s and the Java Virtual Machine. The complete and current documentation for the CIL VM is available at the ECMA[5] website (*https://www.ecma-international.org/publications/standards/Ecma-335.htm*). Search online for "ECMA 335 CIL standard" to find the latest version of this document.

As is the case with all stack machines, CLR VM programs use an evaluation stack for arithmetic operations, passing arguments to functions, and holding local variables and other transient data. Unlike other stack machines, such as the JVM, the size of an element on the top of stack (TOS) is not some specific size like 32 or 64 bits. Instead, the CLR VM operates on items on the TOS as atomic (indivisible) pieces of data. In particular, both 32-bit and 64-bit objects consume one stack entry; the CLR VM doesn't worry about the fact that one object might consume twice as many bits in memory. When performing a 64-bit addition, an application pushes two 64-bit values onto the stack and then adds them, leaving a single 64-bit item on the stack. The CLR VM doesn't treat this any differently from adding two 32-bit values on the stack.

In actual execution, the CLR typically extends smaller values to 32 bits (or some other size) prior to evaluation. However, this is for the benefit of the underlying architecture on which the code runs; ultimately, the CLR must translate CIL into native machine code, and real-world machine architectures tend to force some restrictions onto various operations. For example, some architectures (such as ARM and PowerPC) don't allow you to directly add two 8-bit values. Instead, you must first extend them to 32 bits and then add the 32-bit values together.

4. Some people are confused by the term *virtual machine* and assume that code for a VM must be interpreted. As the CLR does not interpret CIL but instead compiles it to native code, they infer that the CLR is not a VM. However, *virtual machine* literally means "not a real machine." By that definition, the CLR is most certainly a VM even if no interpreter ever executes the code.

5. The European Computer Manufacturer's Association, an international standards organization.

### E.3.1 CLR Primitive Types and Values

The CLR VM supports several primitive (non-object) types, listed in Table E-1. These types are part of the *Common Type System (CTS)*, which is, itself, the basis of the *Common Language Specification (CLS)*. The purpose of the CTS and CLS is to allow interlanguage interoperability. That is, different languages that adhere to the CTS can share data between modules written in those languages.

There are many important data types missing from Table E-1. For example, languages such as C/C++ and C# support the following data types:

| | |
|---|---|
| **`sbyte`** | 8-bit signed integer values |
| **`ushort`** | 16-bit unsigned integer values |
| **`uint`** | 32-bit unsigned integer values |
| **`ulong`** | 64-bit unsigned integer values |

These types are not CLS-compliant. That doesn't mean that the CLR VM cannot handle them, only that if you use them you can't expect those values to be interoperable across different languages (such as ILAsm, C#, C/C++, VB.Net, and F#).

**Table E-1:** CLR Primitive Data Types

| C# type | C++ type | CLS compliant? | ILAsm suffix | Description |
|---|---|---|---|---|
| `bool` | `bool` | Yes | `u1` | Boolean. `0` = `false`, `1` = `true`; also, any nonzero value can also represent `true` in some languages. |
| `byte` | `unsigned char` | Yes | `u1` | Unsigned byte with values in the range `0..255`. Can also represent ASCII (8-bit) characters. |
| `sbyte` | `signed char` | No | `i1` | Signed byte with values in the range `-128..+127`. Can also represent ASCII (7-bit) characters. |
| `ushort` | `unsigned short` | No | `u2` | Unsigned 16-bit integer representing values in the range `0..65535`. |
| `short` | `signed int` | Yes | `i2` | Signed 16-bit integer representing values in the range `-32768..+32767`. |
| `char` | `wchar` | Yes | `u2` | 16-bit Unicode character values (Basic Multilingual Plane [BMP] characters). |
| `uint` | `unsigned int` | No | `u4` | Unsigned 32-bit integers representing values in the range `0..4294967295`. |
| `int` | `int` | Yes | `i4` | Signed 32-bit integers representing values in the range `-2147483348..+2147483347`. |
| `ulong` | `unsigned long`[6] | No | `u4` | Unsigned 64-bit integers representing values in the range `0..18446744073709551615`. |

*(continued)*

6. On 64-bit architectures where C/C++ supports 64-bit unsigned integers.

**Table E-1:** CLR Primitive Data Types (continued)

| C# type | C++ type | CLS compliant? | ILAsm suffix | Description |
|---|---|---|---|---|
| `long` | `long int`[7] | Yes | `i4` | Signed 64-bit integers representing values in the range `-9223372036854775808..` `+9223372036854775807`. |
| `single` | `single` | Yes | `.r4` | 32-bit single-precision floating-point values. |
| `double` | `double` | Yes | `.r8` | 64-bit double-precision floating-point values. |

## E.3.2 CLR Reference and Object Types

In addition to the primitive types, the CLR VM supports *reference types*: pointers to other data values. In languages such as C# and VB.NET, reference types are *managed pointers* that point to class instances (objects), arrays, and strings. In lower-level languages (such as C/C++ and ILAsm), a reference is a machine address of some data value (which could also be an object, array, or string). In CLR terminology, these are *unmanaged pointers.*

Managed pointers are *opaque data types*; an application obtains a managed pointer by requesting a reference for a known object. The application can *dereference* that value (that is, access the data the managed pointer references) and compare it for equality, but it can't otherwise modify the managed pointer. Because a managed pointer is an opaque data type, the application doesn't know anything about its value. It could be a machine address, an index into a lookup table, or any other value that the system can use to reference the actual data object.

Unmanaged pointers, on the other hand, are machine addresses that the application treats as such. An application can add or subtract an integer with an unmanaged pointer (possibly pointing at some illegal address in memory) or otherwise manipulate the pointer as though it were an integer value. Obviously, such manipulation can result in erroneous calculations in an application. This is why languages like VB.NET and C# (attempt to) disallow the use of unmanaged pointers.

## E.3.3 CLR VM Memory Spaces (Homes)

The CLR VM attaches additional semantic information to data it controls so that the JIT compiler knows which items are function arguments, which are local variables, which are static (global) variables, and which are constants. Effectively, the CLR VM associates these different types of memory accesses with their own *memory spaces* (which the CLR VM calls memory *homes*). Memory homes help the CLR VM generate more optimal code.

In a typical VM stack architecture, the CPU references data in three basic locations: in global memory (certain constants, static variables, and objects), on the stack (that is, relative to where the stack pointer register points in memory), and in constant values embedded in machine instructions. In the real world, modern machines provide a large number of registers for

7. On 64-bit architectures where C/C++ supports 64-bit unsigned integers.

manipulating data and may not even have a hardware stack. Mapping data locations in a VM to real hardware can produce suboptimal code.

For example, a Java VM compiler pushes function arguments onto the stack, and the function retrieves those arguments from memory by indexing off the stack pointer register. Native code compilers try to avoid accessing memory by passing function arguments in general-purpose machine registers. Unfortunately, the Java VM doesn't really differentiate between function arguments on the stack, local variables on the stack, temporary calculations on the stack, and so on. As a result, it's difficult for the Java JIT compiler to recognize function arguments and generate code that specifically passes these arguments in registers rather than in memory (on the stack).

To differentiate the memory homes, the CLR VM uses different load instructions to push different types of data onto the CLR VM stack. Table E-2 shows the basic forms of these load instructions.

**Table E-2:** Constant Load Instructions

| Instruction | Description |
|---|---|
| `ldc` | Load constant. Pushes a constant value onto the stack (from a *constant pool* database that the CLR VM maintains). Memory home: constant pool. |
| `ldstr` | Load string. Pushes the address of (reference to) a literal string object onto the stack. The string data itself is in the CLR VM string constant pool. Memory home: constant pool. |
| `ldarg` | Load argument. Pushes a value held in a function argument onto the stack. Note that a compiler does not emit this instruction to create the argument in the first place; rather, it pushes arguments onto the stack in an appropriate order to create those arguments, and the called function uses this instruction to fetch an argument passed to it for calculations in the function's body. Memory home: function arguments. |
| `ldloc` | Load local. Pushes the current value held in a local variable onto the stack. Memory home: function local variables. |
| `ldfld` | Load field. Pushes the current value of a field of some (nonstatic) object. Memory home: heap (dynamically allocated memory). |
| `ldsfld` | Load static field. Pushes the current value of a field of some static class, interface, or module. Memory home: static memory. |
| `ldind`, `ldobj` | Load indirect, load object. Push the value of some variable onto the stack based on an address that was previously on the stack. Memory home: heap (dynamically allocated memory) or arbitrary object in memory (when using unmanaged pointers). |
| `ldelem` | Load element. Pushes the current value of some element of an array. Note that the address of the array and the index into the array are on the stack prior to the execution of this instruction. Memory home: typically the heap, though it could be anything when using unmanaged pointers. |

When the CLR JIT compiler sees a `ldarg` instruction, it knows that the instruction is referencing the function argument's memory home. Based on the underlying CPU architecture, the JIT might emit code to obtain the data from a machine register rather than from stack memory.

## E.4 CLR VM Typed Operands

A typical CPU architecture operates on generic bit data. For example, the x86 `mov` instruction copies blocks of 8, 16, 32, or 64 bits. The CPU doesn't know (or even care) that these bits might represent signed integers, unsigned integers, characters, or some other data type. An x86 `mov al, byteVar` instruction doesn't care what type of data is present in the `byteVar` variable. Other than its size, the CPU doesn't know anything about `byteVar`'s type. The interpretation of the type is left to the application program. CIL memory variables, by contrast, maintain their type information as part of the data, and the CLR JIT compiler uses that information to direct native code generation.

For example, there are actually six different encodings for the `ldarg` instruction (see Table E-3), specifying which argument to load, but none of them encodes type information. Instead, the CIL code maintains type information for all variables (arguments, locals, statics, and other objects), and the argument number in the `ldarg` instruction is an index into a small database maintained for each function.

**Table E-3:** `Ldarg` Instruction Encoding

| ILAsm syntax | Encoding (hexadecimal) | Description |
|---|---|---|
| `ldarg.0` | `06` | Load argument number `0` onto the stack. |
| `ldarg.1` | `07` | Load argument number `1` onto the stack. |
| `ldarg.2` | `08` | Load argument number `2` onto the stack. |
| `ldarg.3` | `09` | Load argument number `3` onto the stack. |
| `ldarg.s` *`index8`* | `11` *`unsigned byte`* | Load argument number `4..255` onto the stack. |
| `ldarg` *`index16`* | `FE 0C` *`unsigned word`* | Load argument number `256..65535` onto the stack. |

Consider the following C# function definition:

```
static void SomeFunc(short shortIntParm, int intParm, string strParm)
{
      << function body >>
}
```

In this example, argument 1 refers to `shortIntParm`, argument 2 refers to `intParm`, and argument 3 refers to `strParm`.[8] These arguments are loaded onto the stack using instructions `ldarg.1`, `ldarg.2`, and `ldarg.3`. The `ldarg.1` instruction will sign-extend (to 32 bits) the 16-bit value associated with argument `shortIntParm`. The `ldarg.2` instruction will push the 32-bit signed integer value associated with `intParm` onto the stack.[9] Finally, the `ldarg.3`

8. In C#, argument `0` generally refers to the `this` pointer, which is typically nil for static functions. However, the exact argument number assignment is chosen by a particular compiler.

9. This assumes that the native `int` size is a 32-bit integer on the given architecture.

instruction will push a reference (an address) of the string object strParm onto the stack (this will be a 32-bit or 64-bit pointer, depending on the underlying CPU architecture).

The ldarg.1 instruction in the *SomeFunc*() function uses the value 1 as an index into a variable table associated with the arguments for SomeFunc(). That table entry tells the JIT compiler that the argument is a 16-bit signed integer (that table entry will likely contain other information that's useful to the JIT compiler, such as where in memory, or in a register, that 16-bit signed value can be found). From this, the JIT knows that it must emit the appropriate code to sign-extend the 16-bit value to 32 bits prior to pushing the value onto the stack or putting it in some other location where the CPU can use the 32-bit integer in a computation.

## E.5 Types and Operations on the Stack

To determine when it must sign-extend (or otherwise change the type) of some value it is manipulating on the stack, the JIT compiler needs to consider the context surrounding the current instruction. Specifically, it must study how the data flows through the program (using a compiler technique known as *data flow analysis*) in order to infer the type changes that will occur.

Consider the following C# code sequence that appears inside some function:

```
int i = 1;
int j = 2;
int k;

k = i + j;
```

The assignment statement might produce the following ISAsm code:

```
ldloc.0         // Push i's value onto stack
ldloc.1         // Push j's value onto stack
add             // Add i + j and leave sum on stack
stloc.2         // Pop stack and store result in k
```

The ldloc instruction is similar in operation to the ldarg instruction, except it loads the value of a local variable rather than a function argument onto the stack. This sequence assumes that i is local variable 0, j is local variable 1, and k is local variable 2.

Now consider the following C# code sequence inside a different function:

```
double i = 1.0;
double j = 2.0;
double k;

k = i + j;
```

Here's the ILAsm a C# compiler might emit for the assignment statement:

```
ldloc.0         // Push i's value onto stack
ldloc.1         // Push j's value onto stack
add             // Add i + j and leave sum on stack
stloc.2         // Pop stack and store result in k
```

In these two examples, the JIT compiler looks at the last two instructions that push data onto the TOS and infers the type of the stack operands by looking up the types of the values that the `ldloc` instructions pushed on the stack. From these instructions, the JIT compiler can determine whether it needs to emit code that adds a pair of 32-bit signed integers or a pair of 64-bit floating-point numbers.

If the data types are incompatible (for example, `ldloc.1` pushes a string reference and `ldloc.2` pushes a 32-bit signed integer), the CLR VM would recognize the type incompatibility and report an appropriate error during JIT compilation.

## E.6 Data Type Conversions

When you do want to work with two incompatible types, a type conversion can resolve the issue. The CLR VM provides two type conversion mechanisms: implicit conversions (which load instructions handle automatically) and explicit type conversions.

Implicit type conversions occur when memory variables and arguments are expanded to the size of the evaluation stack. On most CPUs the evaluation stack holds 32-bit or 64-bit integer values. Therefore, when loading smaller integers onto the stack with `ldloc`, `ldarg`, or similar instructions, the CLR VM automatically sign- or zero-extends 8-bit and 16-bit integer values to 32 bits.[10]

The CLR VM computes floating-point results using double-precision (or better) arithmetic. Therefore, when loading 32-bit single-precision floating-point values onto the stack, the CLR VM implicitly converts them to double-precision. On some architectures, the CLR VM might even convert single- and double-precision floating-point values to something larger than 64 bits (for example, to 80-bit extended precision on the x86).

Not all conversions can be implicit in the CIL, however. For example, if a program attempts to add an integer and a floating-point value together, the two operands must be the same type prior to the addition. Many programming languages will perform an implicit (in the HLL) conversion from integer to floating-point. However, as the CLR VM doesn't perform this conversion implicitly, the HLL compiler must emit an explicit instruction to convert the integer value pushed on the stack to a floating-point value. The `conv.`*`xx`* instructions accomplish this by taking the value on the TOS (whose type the CLR VM will infer via data flow analysis) and

10. On some architectures, the stack holds 64-bit integers only, so the CLR VM will extend 8-, 16-, and 32-bit operands to 64 bits. However, such architectures aren't that common.

converting it to the type specified by the *.xx* instruction suffix. The legal suffixes appear in Table E-4.

**Table E-4:** conv Instruction Suffixes

| ILAsm syntax | Description |
|---|---|
| `conv.i` | Convert integer on TOS to a natural integer (the natural size for the CPU, which could be 16, 32, or 64 bits). |
| `conv.i1` | Convert TOS to a 1-byte signed integer. |
| `conv.i2` | Convert TOS to a 2-byte signed integer. |
| `conv.i4` | Convert TOS to a 4-byte signed integer. |
| `conv.i8` | Convert TOS to an 8-byte signed integer. |
| `conv.u` | Convert integer on TOS to a natural unsigned integer (the natural size for the CPU, which could be 16, 32, or 64 bits). |
| `conv.u1` | Convert TOS to a 1-byte unsigned integer. |
| `conv.u2` | Convert TOS to a 2-byte unsigned integer. |
| `conv.u4` | Convert TOS to a 4-byte unsigned integer. |
| `conv.u8` | Convert TOS to an 8-byte unsigned integer. |
| `conv.r4` | Convert TOS to a 4-byte single-precision floating-point value. |
| `conv.r8` | Convert TOS to an 8-byte double-precision floating-point value. |

The converted result is then pushed back onto the stack (after possible conversion to the stack format). This produces some less-than-intuitive results. For example, if you have a double-precision floating-point value on the stack, executing the `conv.r4` instruction will convert the double-precision value to single and then back to double-precision before pushing it back onto the stack. The value left on the TOS, even though it is in double-precision form, only maintains the precision of a single-precision value. If you were to store this converted value into a single-precision variable and then load it back onto the stack, you would have the exact same bit pattern on the stack as you had immediately after the conversion.

When converting from larger to smaller integer forms, the CLR VM might truncate the value to the number of bits the instruction specifies (for example, `conv.i8` truncates the result to 8 bits). After the truncation, the `conv` instruction either sign-extends (for signed integer conversions) or zero-extends (for unsigned results) the value back to the natural stack size (probably 32 bits, though it could be 64).

In general, conversion from a smaller to a larger integer form is implicit in the load instruction. That is, if the native stack size is 32 bits, loading an 8-bit or 16-bit integer from memory automatically zero- or sign-extends the value to 32 bits (as appropriate for unsigned or signed integers, respectively). To convert an integer to a size larger than the native stack size (for example, from some size to 64 bits on most architectures), you need an explicit `conv.i8` or `conv.u8` instruction.

When converting from an integer form to a floating-point format, you might lose some precision in the conversion based on the size of the floating-point result you produce. You can't convert all possible 32-bit signed integers to a 32-bit single-precision floating-point value without some loss of precision (32-bit floating-point values maintain only 24 bits of precision, whereas 32-bit signed integers require 31 bits of precision). Likewise, you can't convert all 64-bit integers to double-precision, because the double-precision format supports only a 56-bit mantissa. If you attempt to convert a really large integer result to a 32-bit single-precision value, the result may be not-a-number (`NaN`).

When converting real values to integers, the CLR VM truncates the result toward `0`. However, if the real result is too large to fit in the specified integer size, the result is undefined.

Some conversions aren't possible given the source type and value and the destination type. For example, you can't convert the 16-bit unsigned integer value `512` to an 8-bit unsigned integer value, because `512` simply doesn't fit into 8 bits. If you attempt to do this conversion using the `conv.u8` instruction, the result will be `0` (because `conv.u8` will truncate all but the LO 8 bits). This conversion occurs without any indication of error, which can produce defects in the running application if it doesn't expect this behavior. To solve this problem, the CIL instruction set provides `conv.ovf.`*`xx`* instructions for all the integers (signed and unsigned) that check for overflow during the conversion. For example, `conv.ovf.i8` will generate a runtime error if the result on the TOS cannot be converted to a value in the range `-128` through `127`.

## E.7 Basic CLR VM Control Flow

The CLR VM supports a set of control transfer instructions that are quite similar to the JVM instructions. There are conditional branches, unconditional branches, instructions that invoke methods (functions), and special instructions for handling `switch`/`case` statements and exception-handling blocks.

There are two unconditional branch instructions: `br` *`label`* and `br.s` *`label`*. Both instructions transfer control to the statement associated with the *`label`* operand (these instructions are comparable to a `goto` statement in an HLL). The difference between the two instructions is their encoding. The `br.s` instruction is 2 bytes long: a `0x2B` opcode followed by an 8-bit signed displacement. The `br` instruction is 5 bytes long: a `0x38` opcode followed by a 32-bit signed displacement. The (signed) displacement provides the target address (by adding the displacement to the address of the `br` or `br.s` instruction in the CIL object code). Because the `br.s` instruction has only a 1-byte displacement value, it can transfer control only to a destination that is within –128 to +127 bytes of the current instruction. The 5-byte form of the branch instruction allows control transfer to anywhere within a ±2GB range around the instruction. Note that branch targets must appear with the method containing the branch, so a 2GB range is far more than sufficient for any reasonable application (it would be hard to imagine a single method containing 2GB of CIL code).

The most basic conditional branch instructions are `brtrue` and `brfalse`.[11] These two instructions expect a value on the TOS containing the value `true` (any nonzero value) or `false` (`0`). They pop the value off the TOS, compare it against `true` (`brtrue`) or `false` (`brfalse`), and branch to the label specified by the instruction's operand if the TOS value matches the instruction.

An application can use normal arithmetic operations or a comparison instruction to compare two values on the TOS and generate a Boolean result to leave on the TOS for use by the `brtrue` or `brfalse` instructions.

## E.7.1 Comparison Instructions

There are five comparison instructions, listed in Table E-5, that compare the next-on-stack (NOS) value (the one immediately below the TOS) against the TOS value. When comparing values, NOS is the *left* operand of the comparison, and TOS is the *right* operand (that is, NOS *op* TOS; for example, NOS < TOS). These comparison instructions pop two operands from the stack and push a single Boolean value back onto the stack.

**Table E-5:** Comparison Instructions

| ILAsm syntax | Description |
|---|---|
| `ceq` | Compare operands on stack for equality. Push `true` if they are equal, and `false` otherwise. |
| `cgt` | Compare for NOS > TOS and push `true`/`false` based on the result (signed integer and floating-point values). |
| `cgt.un` | Compare for NOS > TOS and push `true`/`false` based on the result (unsigned integer values). |
| `clt` | Compare for NOS < TOS and push `true`/`false` based on the result (signed integer and floating-point values). |
| `clt.un` | Compare for NOS < TOS and push `true`/`false` based on the result (unsigned integer values). |

For unsigned integer operands, compilers typically use the `clt.un` and `cgt.un` instructions. For signed integer operands, compilers typically use `clt` and `cgt`.

Missing from the list of comparisons are tests for not equal, greater than or equal, and less than or equal. However, it's easy enough to synthesize these comparisons: if we invert the result of `ceq`, we effectively get `cne`. Likewise, inverting the result of `clt` produces `cge`, and inverting the result of `cgt` produces `cle`. You can invert a Boolean value on the stack by pushing `0` and comparing the two values (original Boolean value and `0`) for equality using `ceq`.[12]

11. All conditional branches have a long and a short form. Therefore, there are also `brtrue.s` and `brfalse.s` instructions. Henceforth, this chapter will not bother listing the short versions. Just remember that short versions are also available.

12. Another option is to push the value `1` and execute the `xor` instruction. However, this works properly only if the TOS previously contains `0` or `1` (and no other value).

Floating-point comparisons present a special challenge. For normal floating-point values, the `clt` and `cgt` instructions work as you would expect. However, for special floating-point values (specifically, `NaN`s), the `ceq`, `clt`, and `cgt` instructions always return `false` if either operand (or both) is `NaN`—that is, *unordered*. As a general rule, any comparison involving `NaN` should result in `false`, so things are good so far. The problem occurs when you attempt to compute not equal, less than or equal, or greater than or equal. In these situations, you'd also like to leave `false` on the TOS after the comparison if either operand (or both) is unordered.

The `clt.un` and `cgt.un` instructions solve this problem. They return `true` or `false` based on the result of the comparison if both operands are ordered floating-point values; these instructions always push `true` (`1`) if the floating-point values are unordered. In other words, their behavior for unordered values is exactly the opposite of the unadorned `clt` and `cgt` instructions (which always push `false` if the floating-point operands are unordered). Therefore, for floating-point less than or equal or greater than or equal comparisons, use the `cgt.un` or `clt.un` (respectively) instructions and invert the result. The ECMA-335 documentation refers to the `clt.un` and `cgt.un` instructions as *compare less than unsigned or unordered* and *compare greater than unsigned or unordered*. The `.un` suffix stands for *unsigned* or *unordered* depending on whether it is operating on unsigned integers or floating-point values that may be unordered.

There's no `ceq.un` instruction, because the plain `ceq` instruction always returns `false` if either (or both) of the operands is unordered. Two unordered operands are always not equal to each other. Therefore, inverting the result from a plain `ceq` instruction will produce the correct value on the TOS even if the operands are floating-point and unordered.

If you follow the execution of one of the comparison instructions with a `brtrue` or `brfalse` instruction, you can transfer control to some label based on the result of that comparison. Consider the following examples:

```
ldloc.1
ldloc.2
ceq
brtrue TheyreEqual // Branch if local.1 == local.2
  .
  .
  .
ldloc 10
ldloc.3
clt
brtrue LessThan    // Branch if local.10 < local.3
```

To test for less than or equal or greater than or equal, use the `brfalse` instruction to branch if the TOS contains `false` rather than `true`:

```
ldloc.1
ldloc.2
ceq
brfalse TheyreNotEqual // Branch if local.1 != local.2
```

```
  .
  .
  .
ldloc 10
ldloc.3
clt
brfalse LessThan       // Branch if local.10 >= local.3
```

### E.7.2 Branching Instructions

As it turns out, you don't need the comparison instructions at all for most simple compare-and-branch operations. The CIL instruction set includes a set of instructions that combine the compare and `brtrue`/`brfalse` instructions (see Table E-6).

**Table E-6:** Conditional Branch Instructions

| ILAsm syntax | Description |
|---|---|
| `beq` *`label`*<br>`beq.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS == TOS. (No branch if unordered.) |
| `bge` *`label`*<br>`bge.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS >= TOS. (According to ECMA-335, no branch if unordered operands). |
| `bge.un` *`label`*<br>`bge.un.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS >= TOS (unsigned integer operands). |
| `bgt` *`label`*<br>`bgt.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS > TOS. (According to ECMA-335, no branch if unordered operands.) |
| `bgt.un` *`label`*<br>`bgt.un.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS > TOS (unsigned integer operands). |
| `ble` *`label`*<br>`ble.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS <= TOS. (According to ECMA-335, no branch if unordered operands.) |
| `ble.un` *`label`*<br>`ble.un.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS <= TOS (unsigned integer operands). |
| `blt` *`label`*<br>`blt.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS < TOS. (According to ECMA-335, no branch if unordered operands.) |
| `blt.un` *`label`*<br>`blt.un.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS < TOS (unsigned integer operands). |
| `bne` *`label`*<br>`bne.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS != TOS (branch if unordered operands). |
| `bne.un` *`label`*<br>`bne.un.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS != TOS (unsigned operands). |

In addition to all the branch instructions, there is one `jmp` instruction. This instruction contains a single method (function) name as an operand and transfers control from one method to another. This is a special instruction that compilers won't normally generate for user-written statements, so you won't see `jmp` very often in compiler output.

The `switch` instruction, which implements a table-driven selection statement in an HLL, consists of the keyword `switch` followed by a table of labels. This instruction expects a single unsigned integer on the stack. If the value is in the range `0` through ***`number of table elements`*** `-1`, the `switch` instruction transfers control to the label at the value's index into the table. If the value is outside this range, the control transfers to the first CIL instruction following the table. The binary encoding of the `switch` instruction is the opcode `0x45`, followed by a 32-bit unsigned integer specifying the number of table entries, and then the table with each entry containing a 32-bit signed integer value (which is the offset to the corresponding label from the byte immediately following the `switch` instruction).

There are also `call` and `callvirt` instructions for calling static and virtual methods, respectively. These instructions take the method name as their argument. The `callvirt` instruction also expects a pointer to the corresponding object on the TOS. Both instructions expect the caller to push all parameter arguments, in the order of their declaration, on the stack prior to the execution of the call.

## E.8 The CIL Instruction Set

Table E-7 lists all of the CIL instructions and provides a brief discussion of their function. The intent of this appendix is not to teach you how to write IL assembly language, only to prepare you to read dumps of ILAsm from compiled programs. Thus, treat this chapter as a reference guide when reading compiler output in order to figure out how a given HLL generates code for its statements.

**Table E-7:** CIL Instructions

| Instruction | Description |
|---|---|
| `add` | Adds TOS and NOS, leaving result on TOS. |
| `add.ovf` | Adds TOS and NOS (signed integers only), leaving result on TOS. Raises an exception if an unsigned overflow occurs. |
| `add.ovf.un` | Adds TOS and NOS (unsigned integers only), leaving result on TOS. Raises an exception if an unsigned overflow occurs. |
| `and` | Computes bitwise logical AND of TOS and NOS, leaving result on TOS. |
| `arglist` | Pushes the address of a C/C++ `varargs` argument list. |
| `beq` *`label`*<br>`beq.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS <= TOS. (According to ECMA-335, no branch if unordered operands.) |
| `bge` *`label`*<br>`bge.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS >= TOS. (According to ECMA-335, no branch if unordered operands.) |
| `bge.un` *`label`*<br>`bge.un.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS >= TOS (unsigned integer operands). |
| `bgt` *`label`*<br>`bgt.s` *`label`* | Pop two items from stack and branch to *`label`* if NOS > TOS. (According to ECMA-335, no branch if unordered operands). |

*(continued)*

**Table E-7:** CIL Instructions (continued)

| Instruction | Description |
|---|---|
| `bgt.un` *label*<br>`bgt.un.s` *label* | Pop two items from stack and branch to *label* if NOS > TOS (unsigned integer operands). |
| `ble` *label*<br>`ble.s` *label* | Pop two items from stack and branch to *label* if NOS <= TOS. (According to ECMA-335, no branch if unordered operands.) |
| `ble.un` *label*<br>`ble.un.s` *label* | Pop two items from stack and branch to *label* if NOS <= TOS (unsigned integer operands). |
| `blt` *label*<br>`blt.s` *label* | Pop two items from stack and branch to *label* if NOS < TOS. (According to ECMA-335, no branch if unordered operands.) |
| `blt.un` *label*<br>`blt.un.s` *label* | Pop two items from stack and branch to *label* if NOS < TOS (unsigned integer operands). |
| `bne` *label*<br>`bne.s` *label* | Pop two items from stack and branch to *label* if NOS != TOS (branch if unordered operands). |
| `bne.un` *label*<br>`bne.un.s` *label* | Pop two items from stack and branch to *label* if NOS != TOS (unsigned operands). |
| `box` *type* | Type coercion from a value to a system object on the heap. Converts a value to an object for compatibility with object-oriented programming. |
| `br` *label*<br>`br.s` *label* | Transfer control to specified *label* (in current function). |
| `break` | Breakpoint instruction. |
| `brfalse` *label*<br>`brfalse.s` *label* | Pops Boolean value and branches to target *label* if TOS contained `0`. |
| `brinst` *label*<br>`brinst.s` *label* | Branch to *label* if TOS is a non-`null` pointer (synonym for `brtrue`). |
| `brNull` *label*<br>`brNull.s` *label* | Branch if TOS contains `null` (synonym for `brfalse`). |
| `brtrue` *label*<br>`brtrue.s` *label* | Pops TOS and branches to *label* if value is nonzero. |
| `brzero` *label*<br>`brzero.s` *label* | Synonym for `brfalse`. |
| `call` *method* | Calls (static) method specified by operand. Caller pushes arguments in order prior to instruction. |
| `calli`<br>*methodDescr* | Calls method whose address is on TOS (with arguments specified by descriptor argument). Caller pushes arguments in order prior to pushing address and executing instruction. |
| `callvirt`<br>*method* | Call method specified by argument. Caller pushes arguments in order prior to executing instruction. First argument must be pointer to object associated with virtual method. |
| `castclass` *obj* | Cast object to class specified by argument. |
| `ceq` | Compare NOS to TOS for equality. Leave Boolean result on TOS. |
| `cgt` | Compare NOS to TOS for greater than. Leave Boolean result on TOS. Unordered operands always produce `false`. |

*(continued)*

**Table E-7:** CIL Instructions (continued)

| Instruction | Description |
|---|---|
| `cgt.un` | Compare unsigned or floating-point NOS to TOS for greater than. Leave Boolean result on TOS. Unordered operands always produce `true`. |
| `ckfinite` | Throw an exception if the TOS is not a finite (floating-point) value. |
| `clt` | Compare NOS to TOS for less than. Leave Boolean result on TOS. Unordered operands always produce `false`. |
| `clt.un` | Compare unsigned or floating-point NOS to TOS for less than. Leave Boolean result on TOS. Unordered operands always produce `true`. |
| `constrained .type` | This is a prefix for the `callvirt` instruction. It allows value and reference types to be used interchangeably for certain method calls. |
| `conv.i`<br>`conv.i1`<br>`conv.i2`<br>`conv.i4`<br>`conv.i8` | Converts operand on TOS to the native signed integer format (`.i`) or to a specified integer size (`.1` through `.i8`). |
| `conv.ovf.i`<br>`conv.ovf.i1`<br>`conv.ovf.i2`<br>`conv.ovf.i4`<br>`conv.ovf.i8` | Converts operand on TOS to the native signed integer format (`.i`) or to a specified integer size (`.i1` through `.i8`). Raises an overflow exception if the original value on TOS cannot be converted to the specified size. |
| `conv.ovf.i.un`<br>`conv.ovf.i1.un`<br>`conv.ovf.i2.un`<br>`conv.ovf.i4.un`<br>`conv.ovf.i8.un` | Converts unsigned operand on TOS to the native signed integer format (`.i`) or to a specified integer size (`.i1` through `.i8`). Raises an overflow exception if the original value on TOS cannot be converted to the specified size. |
| `conv.ovf.u`<br>`conv.ovf.u1`<br>`conv.ovf.u2`<br>`conv.ovf.u4`<br>`conv.ovf.u8` | Converts operand on TOS to the native unsigned integer format (`.u`) or to a specified unsigned integer size (`.u1` through `.u8`). Raises an exception if overflow occurs. |
| `conv.ovf.u.un`<br>`conv.ovf.u1.un`<br>`conv.ovf.u2.un`<br>`conv.ovf.u4.un`<br>`conv.ovf.u8.un` | Converts unsigned operand on TOS to the native unsigned integer format (`.u`) or to a specified unsigned integer size (`.u1` through `.u8`). Raises an exception if overflow occurs. |
| `conv.r.un` | Converts unsigned operand on TOS to a floating-point value. |
| `conv.r4` | Converts TOS to a 32-bit floating-point value (which then gets converted to the native floating-point format on TOS). |
| `conv.r8` | Converts TOS to a 64-bit floating-point value (which then gets converted to the native floating-point format on TOS). |
| `conv.u`<br>`conv.u1`<br>`conv.u2`<br>`conv.u4`<br>`conv.u8` | Converts operand on TOS to the native unsigned integer format (`.u`) or to a specified unsigned integer size (`.u1` through `.u8`). |
| `cpblk` | Copy a block of memory from one location to another. Source address, destination address, and size are all on the stack. |

*(continued)*

**Table E-7:** CIL Instructions (continued)

| Instruction | Description |
|---|---|
| `cpobj` *`type`* | Copies the data of some source object (address on stack, of type *type*) to a destination object (whose address is also on the stack). |
| `div` | Divides NOS by value on TOS, leaving result on TOS (signed integer or floating-point values). |
| `div.un` | Divides NOS by value on TOS, leaving result on TOS (unsigned integer values). |
| `dup` | Duplicates the value on TOS. |
| `endfault` | Ends the `fault` clause of an exception block. |
| `endfilter` | Ends the `filter` clause of an exception block. |
| `endfinally` | Ends the `finally` clause of an exception block. |
| `initblk` | Initializes a block of memory to a given value. |
| `initobj` *`type`* | Initializes an object (address on TOS) of type *type* with appropriate initial values (`0`s or `null`s). |
| `isinst` *`type`* | TOS contains an object reference (address). Pushes `true` if the object of type *type*, and `false` otherwise. |
| `jmp method` | Exits the current method and transfers control to the method specified by the operand. |
| `ldarg` *`index32`*<br>`ldarg.s` *`index8`*<br>`ldarg.0`<br>`ldarg.1`<br>`ldarg.2`<br>`ldarg.3` | Loads the method/function argument specified by the operand *index* onto the TOS (or index `0`, `1`, `2`, or `3` for the short forms). |
| `ldc.i4`<br>`int_const32`<br>`ldc.i4.s`<br>`int_const8` | Loads the specified 32-bit signed constant onto the TOS (`ldc.i4.s` is a special short form that supports an 8-bit signed integer constant). |
| `ldc.i4.m1`<br>`ldc.i4.0`<br>`ldc.i4.1`<br>`ldc.i4.2`<br>`ldc.i4.3`<br>`ldc.i4.4`<br>`ldc.i4.5`<br>`ldc.i4.6`<br>`ldc.i4.7`<br>`ldc.i4.8` | Compact form of `ldc`. Pushes the value `-1..8` onto the TOS. |
| `ldc.r4`<br>`flt_const32`<br>`ldc.r8`<br>`flt_const64` | Loads the specified floating-point constant onto the TOS. |
| `ldelem` *`type`* | TOS contains an array reference and an index. Array elements have type *type*. This instruction loads the specified array element onto the TOS. |

*(continued)*

**Table E-7:** CIL Instructions (continued)

| Instruction | Description |
|---|---|
| `ldelem.i`<br>`ldelem.i1`<br>`ldelem.i2`<br>`ldelem.i4`<br>`ldelem.i8` | TOS contains an array reference and an index. Array elements are a native integer, 1-byte integer, 2-byte integer, and so on, as specified by instruction suffix. This instruction loads the specified array element onto the TOS. |
| `ldelem.r4`<br>`ldelem.r8` | TOS contains an array reference and an index. Array elements are single- or double-precision floating-point values, as specified by instruction suffix. This instruction loads the specified array element onto the TOS. |
| `ldelem.ref` | TOS contains an object reference and an index. Array elements are some object type. This instruction loads the specified array element onto the TOS. |
| `ldelem.u1`<br>`ldelem.u2`<br>`ldelem.u4`<br>`ldelem.u8` | TOS contains an array reference and an index. Array elements are a 1-byte unsigned integer, 2-byte unsigned integer, and so on, as specified by instruction suffix. This instruction loads the specified array element onto the TOS. |
| `ldelema` *`class`* | The stack contains an object reference (of type *`class`*) and an index. Loads the address of the specified element onto TOS. |
| `ldfld` *`field`* | TOS contains an object reference (address). Loads the field specified by the operand onto the TOS. |
| `ldflda` *`field`* | TOS contains an object reference (address). Loads the address of the field specified by the operand onto the TOS. |
| `ldftn` *`method`* | Loads the address of the method whose name is specified by the operand onto the stack. |
| `ldind.i`<br>`ldind.i1`<br>`ldind.i2`<br>`ldind.i4`<br>`ldind.i8` | The TOS contains the address of a signed integer (natural, 1-byte, 2-byte, 4-byte, or 8-byte). This instruction loads the value of that integer onto the TOS. |
| `ldind.r4`<br>`ldind.r8` | The TOS contains the address of a floating-point value (32-bit or 64-bit). This instruction loads the real value onto the TOS. |
| `ldind.ref` | The TOS contains the address of an object. This instruction loads the value of that object onto the TOS. |
| `ldind.u1`<br>`ldind.u2`<br>`ldind.u4`<br>`ldind.u8` | The TOS contains the address of an unsigned integer (1-byte, 2-byte, 4-byte, or 8-byte). This instruction loads the value of that integer onto the TOS. |
| `ldlen` | The TOS contains an array reference. This instruction loads the size of that array onto the TOS as an unsigned integer. |
| `ldloc` *`index32`*<br>`ldloc.s` *`index8`*<br>`ldloc.0`<br>`ldloc.1`<br>`ldloc.2`<br>`ldloc.3` | This instruction loads the value of a local variable onto the TOS. The local variable is specified by *`index`* as either `0`, `1`, `2`, `3`; an unsigned 8-bit index; or a 32-bit unsigned index. |
| `ldloca` *`index32`*<br>`ldloca.s`<br>*`index8`* | This instruction loads the address of a local variable onto the TOS. The local variable is specified by *`index`* as either an unsigned 8-bit index or a 32-bit unsigned index. |

*(continued)*

**Table E-7:** CIL Instructions (continued)

| Instruction | Description |
|---|---|
| `Ldnull` | Loads the value `null` onto the TOS. |
| `ldobj` *`type`* | TOS contains a reference to an object of type *`type`*. This instruction copies the value of that object onto the TOS. |
| `ldsfld` *`field`* | The operand is the class and field name of a static field. This instruction loads the value of that field onto the TOS. |
| `ldsflda` *`field`* | The operand is the class and field name of a static field. This instruction loads the address of that field onto the TOS. |
| `ldstr` *`string`* | This instruction loads a reference to the string literal *`string`* onto the TOS. Note that in the actual CIL object file, the `ldstr` instruction encodes the string as a 32-bit index, providing a link to the string data appearing elsewhere in the CIL file (the metadata, or string constant database, in the CIL). |
| `ldtoken` *`token`* | This instruction pushes a runtime token onto the stack that represents the *`token`* operand. Metadata is information in the CIL object file containing information (such as runtime type information) about the program. For more information, see *https://en.wikipedia.org/wiki/Metadata_(CLI)*. |
| `ldvirtftn` *`method`* | TOS contains an object reference. This instruction pushes the address of the virtual method specified by that reference and the *`method`* operand onto the TOS. |
| `leave` *`label`*<br>`leave.s` *`label`* | These instructions are functionally equivalent to the `br` and `br.s` instructions except they allow you to exit from an exception-handling block (`br` instructions can transfer control only within the body of an exception handler, never out of it). |
| `localloc` | TOS contains an unsigned integer specifying some number of bytes to allocate on the local heap. This instruction allocates the specified number of bytes on the local heap and returns a pointer to that block on the TOS. When the current function returns, all data on the local heap is released to the system for future use. |
| `mkrefany` *`class`* | Pops a pointer to a block of data off the stack, converts that pointer to a reference of type *`class`*, and pushes the resulting reference back onto the TOS. |
| `mul` | Multiplies NOS by TOS, leaving the product on TOS. No exception if overflow occurs. |
| `mul.ovf` | Multiples two signed integers (NOS by TOS), leaving result on the TOS. If a signed overflow occurs, this instruction raises an exception. |
| `mul.ovf.un` | Multiples two unsigned integers (NOS by TOS), leaving result on the TOS. If an unsigned overflow occurs, this instruction raises an exception. |
| neg | Negates the value on TOS (floating-point or signed integer). |
| `newarr` *`type`* | TOS contains an unsigned integer specifying the number of elements. This instruction allocates storage for a single-dimensional array whose element type is *`type`* and pushes a reference to the array onto the stack. |

*(continued)*

**Table E-7:** CIL Instructions (continued)

| Instruction | Description |
|---|---|
| `newobj` *`constr`* | This instruction expects zero or more parameters on the stack (as appropriate for the class constructor specified by *`constr`*). This instruction allocates sufficient storage for an object of the *`constr`*'s class and then invokes the constructor. |
| `nop` | No operation (does nothing). |
| `not` | Bitwise inversion (complement) of the value on TOS (leaving the result on TOS). Note that pushing `true` and executing `not` does not produce `false`. Instead, this sequence produces a value that has all `1` bits except for bit `0` (which the system will still treat as the value `true` for the purposes of compare and conditional branch instructions). |
| `or` | Bitwise OR of NOS with TOS, leaving the result on TOS. |
| `pop` | Pops (deletes) the value on TOS. |
| `readonly` | Used as a prefix for `ldelema` instruction to specify that the resulting address points at a read-only value in memory. |
| `refanytype` | A value type reference appears on the TOS. This instruction replaces that value with the type token for that value. Used with the `mkrefany` instruction. |
| `refanyval` | A value type reference appears on the TOS. This instruction replaces that value type reference with the address for that value. Used with the `mkrefany` instruction. |
| `rem` | Remainder when dividing signed integer NOS by TOS. |
| `rem.un` | Remainder when dividing unsigned integer NOS by TOS. |
| `ret` | Return from function/method. |
| `rethrow` | Rethrow the current exception. |
| `shl` | Shifts NOS to the left the number of bits specified by TOS and leaves the result on TOS. |
| `shr` | Shifts NOS to the right the number of bits specified by TOS and leaves the result on TOS. Shift is an arithmetic shift right, shifting the sign bit into the next-to-HO bit position of the value. |
| `shr.un` | Shifts NOS to the right the number of bits specified by TOS and leaves the result on TOS. Shift is a logical shift right, shifting `0`s into the HO bit position of the value. |
| `sizeof` *`type`* | Pushes the size of *`type`*, in bytes, onto the TOS as an unsigned integer. |
| `starg` *`index32`*<br>`starg.s` *`index8`* | Pops the value off TOS and stores it into the method/function argument specified by the operand *`index`*. |
| `stelem` *`type`* | TOS contains a value (of type *`type`*); NOS contains an index and the stack entry below NOS contains an array reference. This instruction stores the specified value into the array at the given index. |
| `stelem.i`<br>`stelem.i1`<br>`stelem.i2`<br>`stelem.i4`<br>`stelem.i8` | Top three elements on the stack contain a (signed integer) value, index, and array reference. This instruction stores the value into the appropriate array element. The element type is either a natural signed integer, or a 1-, 2-, 4-, or 8-byte signed integer as specified by the instruction suffix. |

*(continued)*

**Table E-7:** CIL Instructions (continued)

| Instruction | Description |
|---|---|
| `stelem.r4`<br>`stelem.r8` | Top three elements on the stack contain a (floating-point) value, index, and array reference. This instruction stores the value into the appropriate array element. The element type is either a single- or double-precision float as specified by the instruction suffix. |
| `stelem.ref` | Top three elements on the stack contain an object reference value, index, and array reference. This instruction stores the object reference value into the appropriate array element. |
| `stfld` *`field`* | NOS contains an object reference. TOS contains a value. The `stdfld` instruction stores the value into the field of the object specified by the *`field`* operand. |
| `stind.i`<br>`stind.i1`<br>`stind.i2`<br>`stind.i4`<br>`stind.i8` | TOS contains a signed integer value. NOS contains the address of a natural, 1-, 2-, 4-, or 8-byte signed integer (as specified by the instruction suffix). This instruction stores the integer value at the specified address.<br>Note that there are no store indirect unsigned instructions. Compilers might generate these instructions to store unsigned values, as the store operation is identical for signed and unsigned integers. |
| `stind.r4`<br>`stind.r8` | TOS contains a floating-point value. NOS contains the address of a single- or double-precision floating-point variable (as specified by the instruction suffix). This instruction stores the float value at the specified address. |
| `stind.ref` | TOS contains an object value. NOS contains a pointer to an object variable of the same type. This instruction stores the value at the specified address. |
| `stloc index32`<br>`stloc.s index8`<br>`stloc.0`<br>`stloc.1`<br>`stloc.2`<br>`stloc.3` | Pops the value off TOS and stores it into the local variable specified by the operand *`index`* or local variable `0`, `1`, `2`, or `3` (as specified by the instruction suffix). |
| `stobj` *`class`* | An object value of type *`class`* appears on TOS; a pointer to an object of type class appears on NOS. This instruction pops the value and stores it at the specified address. |
| `stsfld`<br>*`class:field`* | An object value of type *`class`* appears on TOS. This instruction pops that value off TOS and stores it into the static field specified by *`class:field`*. |
| `sub` | Computes NOS – TOS and leaves the result on TOS. |
| `sub.ovf` | The stack contains two signed integer values. This instruction computes NOS – TOS and leaves the result on TOS. If a signed arithmetic overflow occurs, this instruction raises an exception. |
| `sub.ovf.un` | The stack contains two unsigned integer values. This instruction computes NOS – TOS and leaves the result on TOS. If an unsigned arithmetic overflow occurs, this instruction raises an exception. |
| `switch [`*`table`*`]` | The *`table`* operand is a list of statement labels in the current function (subject to the same rules as branch labels). This instruction pops an unsigned integer value from the TOS and uses it as an index into the table. If the index is less than the size of the table, this instruction transfers control to the corresponding label. If the index is out of range, control transfers to the next instruction following the `switch` instruction. |

*(continued)*

**Table E-7:** CIL Instructions (continued)

| Instruction | Description |
|---|---|
| `tailcall` | This is a prefix to the `call`, `callvert`, or `calli` instructions. It performs a tail-recursive call. |
| `throw` | TOS contains an object reference (an exception object). This instruction throws an exception. |
| `unaligned` | An instruction prefix that may appear before certain load and store instructions informing the system that unaligned memory accesses are legal. |
| `unbox` *`type`* | Undoes the effects of a box instruction. That is, it takes a boxed object reference and converts it to a value (of type *type*). |
| `unbox.any` *`type`* | Undoes the effects of a box instruction. That is, it takes a boxed object reference and converts it to a value (of type *type*). When unboxing a value type, this is comparable to `unbox` followed by `ldobj`. When unboxing a reference type, this is basically a cast operation. |
| `volatile` | A prefix to various load and store operations informing the system that multiple threads might access this value (thus, the CLR JIT optimizer must not cache values, and multiple stores to the same address cannot be suppressed). |
| `xor` | Bitwise exclusive-OR of NOS with TOS, leaving the result on TOS. |

## E.9 For More Information

Box, Don, with Chris Sells. *Essential .NET, Volume 1.* Boston: Addison-Wesley Professional, 2002.

Lidin, Serge. *Inside Microsoft .NET IL Assembler.* Redmond, WA: Microsoft Press, 2002.

———. *Expert .NET 2.0 IL Assembler.* Berkeley, CA: Apress, 2006.

———. *.NET IL Assembler.* Berkeley, CA: Apress, 2014.

Pratschner, Steven. *Customizing the Microsoft® .NET Framework Common Language Runtime (Developer Reference).* Redmond, WA: Microsoft Press, 2005.

**NOTE** *The resources at* www.writegreatcode.com *contain links to the full documentation for the CIL.*