.NET 10 Common Intermediate Language

M. MOHR

mateusz.mohr@student.wat.edu.pl

Military University of Technology, Faculty of Cybernetics
Kaliskiego Str. 2, 00-908 Warsaw, Poland

The .NET Common Intermediate Language (CIL, IL – Intermediate Language) is a programming language that is an intermediate version between source code and machine code. It is standardized and closely linked to the .NET platform and high-level languages such as C#, F#, and Visual Basic .NET (VB.NET). The .NET platform uses a two-step compilation process. It converts high-level source code (e.g., C#) into a .NET platform assembly containing CIL code in the form of an executable file (.exe) or dynamic link library (.dll). Only at runtime is this intermediate code, loaded by the .NET CLR (Common Language Runtime), translated into native processor instructions appropriate for the specific operating system and hardware architecture. Thanks to this approach, the same CIL code can be run on different platforms such as Windows, Linux, or macOS without having to recompile the source code.
Although intermediate language code is not written directly by the programmer, it can be analyzed and edited using special tools such as ILDASM and ILASM developed by Microsoft, or ILSpy, a tool developed independently of Microsoft as an open-source project. The characteristic two-step compilation process makes it relatively easy to analyze the source code of a program compiled and delivered as an executable file, and even to recreate it in a form similar to the original code.

Keywords: .NET, CIL, Common Intermediate Language.

DOI: 10.5604/01.3001.0055.7729

1. Introduction

.NET is a free, cross-platform, open-source programming platform that enables software development in multiple programming languages, including C#, F#, and VB.NET. The current version of the platform (.NET 10) was released in November 2025. The platform uses a two-step compilation process. The source code prepared by the developer is converted by an appropriate compiler (in the case of C#, this is Roslyn) into CIL code, the current instructions for which are described in the 6th edition of the ECMA-335 standard from June 2012. Then, usually when the program is launched, the intermediate code is loaded by CLR and compiled into native processor instructions, which are executed by the computer [1], [2].

Regardless of the language in which the source code was originally written, all .NET applications use the same intermediate language, allowing a single application to run on many different hardware platforms using different operating systems [1]. One of the tools that allows you to view the CIL code of an application (or library) in text form is ILDASM. With its help, it is possible to export CIL code to a text file, which can be modified and then restored to its previous form using the ILASM tool [2]. Another tool that allows you to analyze .NET applications is ILSpy, which not only allows you to view CIL code, but also to decompile the assembly into high-level code, such as C# [3].

By preparing a simple console application in C# that uses high-level instructions available on the platform since .NET 6, it is possible to observe how the source code is translated into an intermediate language [2]. An application that prints the classic Hello, World message to the console, implemented as a single instruction Console.WriteLine("Hello, World!") can be analyzed with ILSpy after compilation.

By analyzing the intermediate code view (Figure 2) with the help of the ECMA-335 standard documentation, one can become familiar with the basic structure of CIL. The .method declaration defines the Main method, specifying its attributes: access level (private), method name resolution (hidebysig), whether it is static (static), and return type (void). Next, the method name and a list of parameters (string[] args) are provided, which are part of its signature. The content of the method is enclosed in curly brackets.

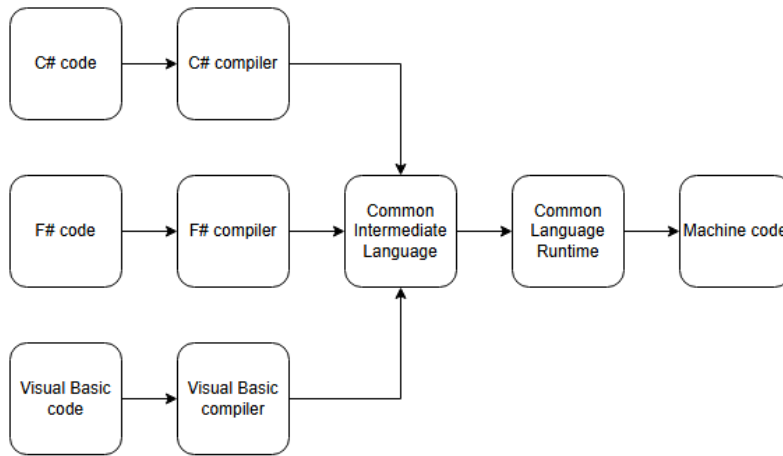

Fig. 1. Process of compiling and running .NET programs
Source: Own work based on [1]

```
<Main>$(string[]) : void
  .method private hidebysig static
      void '<Main>$' (
          string[] args
      ) cil managed
  {
      // Method begins at RVA 0x2050
      // Header size: 1
      // Code size: 12 (0xc)
      .maxstack 8
      .entrypoint

      IL_0000: ldstr "Hello, World!"
      IL_0005: call void [System.Console]System.Console::WriteLine(string)
      IL_000a: nop
      IL_000b: ret
  } // end of method Program::'<Main>$'
```

Fig. 2. CIL code for the Main function in the ILSpy program that prints the string Hello, World! to the console
Source: Own work

```
<Main>$(string[]) : void
 // ConsoleApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
 // Program
 using System;

 private static void <Main>$(string[] args)
 {
     Console.WriteLine("Hello, World!");
 }
```

Fig. 3. C# code converted from CIL code using ILSpy
Source: Own work

The first line after the comments (.maxstack 8) specifies the maximum number of elements on the evaluation stack. The second (.entrypoint) indicates the entry point of the program. The following intermediate language instructions labeled IL_0000, IL_0005, IL_000a, IL_000b represent the operations performed by the program. The ldstr instruction places a string stored in the program metadata on the stack (in fact, the argument of ldstr is an index to the metadata container #US – User String Heap, under which the string Hello, World! is stored). The call instruction invokes System.Console: WriteLine, removing the argument from the stack. The nop instruction does nothing, while ret ends the execution of the method [4], [5].

Using ILSpy, the intermediate code can be converted back to a higher-level language (Figure 3) and analyzed in a way that is easier for humans to understand.

2. CIL structure and syntax

CIL is a language formalized by the ECMA-335 standard, which describes all of its instructions in detail. Understanding its structure and syntax allows you to follow how high-level language instructions are mapped to elements executed by the CLR and to analyze them in detail using the ECMA-355 standard documentation. This chapter presents several simple examples of mapping C# code to CIL, which are used to discuss some of the CIL instructions.

2.1. Subtraction

Code 1. A method performing subtraction in C#

```
public class Calculator
{
    public int Subtraction(int a, int b)
    {
        return a - b;
    }
}
```

Code 2. A method performing subtraction in CIL

```
.method public hidebysig
    instance int32 Subtraction (
        int32 a,
        int32 b
    ) cil managed
{
    // Method begins at RVA 0x20fc
    // Header size: 12
    // Code size: 9 (0x9)
    .maxstack 2
    .locals init (
        [0] int32
    )

    IL_0000: nop
    IL_0001: ldarg.1
    IL_0002: ldarg.2
    IL_0003: sub
    IL_0004: stloc.0
    IL_0005: br.s IL_0007

    IL_0007: ldloc.0
    IL_0008: ret
} // end of method
Calculator::Subtraction
```

The presented CIL code (Code 2) shows the definition of the Substraction method. The attributes of the .method directive specify that it is public and distinguished by its full signature (hidebysig). It is an instance method (instance), which means it operates on a specific object of the class. The int32 appearing before the method name specifies the return type, while int32 a and int32 b inside the parentheses define the method's input parameters. The .maxstack 2 directive specifies the depth of the evaluation stack, and the .local init section declares a local variable in which the result of the subtraction operation is stored [4].

The subsequent instructions performed in the method are presented in the table 1.

Tab. 1. CIL instructions for the Subtraction method

Label	Instruction	Operation
IL_0000	nop	The instruction performs no operation.
IL_0001	ldarg.1	Pushes the value of the first method argument (a) onto the stack.
IL_0002	ldarg.2	Pushes the value of the second method argument (b) onto the stack.
IL_0003	sub	Pops two values from the stack, performs subtraction (a–b), and pushes the result onto the stack.
IL_0004	stloc.0	Stores the result from the stack into the local variable at index 0.
IL_0005	br.s IL_0007	Unconditional branch to instruction IL_0007.
IL_0007	ldloc.0	Pushes the value of the local variable (the subtraction result) onto the stack.
IL_0008	ret	Ends method execution, returning the value at the top of the stack.

Labels in the form IL_0000, IL_0001, etc. indicate the byte offset of a given instruction relative to the beginning of the method.

2.2. Iterating a while loop

Code 3. Method executing a while loop in C#

```
public class Loops
{
    public void WhileLoop(int i)
    {
        while (i < 3)
        {
            Console.WriteLine(i);
            i++;
        }
    }
}
```

Code 4. Method executing a while loop in CIL

```
.method public hidebysig
    instance void WhileLoop (
        int32 i
    ) cil managed
{
    // Method begins at RVA 0x2068
    // Header size: 12
    // Code size: 26 (0x1a)
    .maxstack 2
    .locals init (
        [0] bool
    )

    IL_0000: nop
    IL_0001: br.s IL_0011
    // loop start (head: IL_0011)
        IL_0003: nop
        IL_0004: ldarg.1
        IL_0005: call void
[System.Console]System.Console::WriteLine
(int32)
        IL_000a: nop
        IL_000b: ldarg.1
        IL_000c: ldc.i4.1
        IL_000d: add
        IL_000e: starg.s i
        IL_0010: nop

        IL_0011: ldarg.1
        IL_0012: ldc.i4.3
        IL_0013: clt
        IL_0015: stloc.0
        IL_0016: ldloc.0
        IL_0017: brtrue.s IL_0003
    // end loop

    IL_0019: ret
} // end of method Loops::WhileLoop
```

In the code showing the execution of the while loop, the declaration of the .method directive is distinguished by the return type. The method from the previous example returned a numeric value, while WhileLoop has a void return type because it does not return any result. In this case, the method takes only one input argument. It is worth noting the .locals init section, which, despite the lack of a return value, declares a local variable of type bool, which is used to store the result of the loop condition comparison and is used by the conditional jump statement.

The subsequent instructions performed in the method are presented in the table 2.

Tab. 2. CIL instructions for the WhileLoop method

Label	Instruction	Operation
IL_0000	nop	The instruction performs no operation.
IL_0001	br.s IL_0011	Unconditional branch to the loop condition check.
IL_0003	nop	The instruction performs no operation.
IL_0004	ldarg.1	Pushes the value of argument i onto the stack.
IL_0005	call void [System.Console]System.Console ::WriteLine(int32)	Calls Console.WriteLine (int32), popping i from the stack and printing it to the console.
IL_000a	nop	The instruction performs no operation.
IL_000b	ldarg.1	Loads the current value of i onto the stack.
IL_000c	ldc.i4.1	Pushes the constant value 1 onto the stack.
IL_000d	add	Pops two values from the stack and pushes the result i + 1.
IL_000e	starg.s i	Stores the result from the stack back into argument i.
IL_0010	nop	The instruction performs no operation.
IL_0011	ldarg.1	Loads i onto the stack (start of the condition check).
IL_0012	ldc.i4.3	Pushes the constant value 3 onto the stack.
IL_0013	clt	Compares two values from the stack and pushes 1 (true) if i < 3, otherwise 0.
IL_0015	stloc.0	Stores the comparison result into a local variable.
IL_0016	ldloc.0	Loads the local variable onto the stack.
IL_0017	brtrue.s IL_0003	If the value on the stack is true, branches to the beginning of the loop body (IL_0003).
IL_0019	ret	Ends the execution of the method.

3. Basic CIL instructions

The ECMA-335 standard allows for a more detailed review of CIL instructions. Based on this standard, a table has been prepared containing selected basic instructions, the full list of which can be found in the documentation.

Tab. 3. Selected CIL instructions

Instruction	ECMA-335 Description
add	Add two values, returning a new value.
and	Bitwise AND of two integral values, returns an integral value.
arglist	Return argument list handle for the current method.
beq.<length>	Branch to target if equal.
bge.<length>	Branch to target if greater than or equal to.
bgt.<length>	Branch to target if greater than.
ble.<length>	Branch to target if less than or equal to.
blt.<length>	Branch to target if less than.
br.<length>	Branch to target.
break	Inform a debugger that a breakpoint has been reached.
brfalse.<length>	Branch to target if value is zero (false).
brtrue.<length>	Branch to target if value is non-zero (true).
call	Call method described by method.
ceq	Push 1 (of type int32) if value1 equals value2, else push 0.
cgt	Push 1 (of type int32) if value1 > value2, else push 0.
clt	Push 1 (of type int32) if value1 < value2, else push 0.
conv.<to type>	Data conversion
cpblk	Copy data from memory to memory.
div	Divide two values to return a quotient or floating-point result.
dup	Duplicate the value on the top of the stack.
initblk	Set all bytes in a block of memory to a given byte value.
jmp	Exit current method and jump to the specified method.
ldarg.<length>	Load argument onto the stack.
ldarga.<length>	Load an argument address.
ldc.<type>	Load numeric constant.
ldloc	Load local variable onto the stack.
ldloca.<length>	Load local variable address.
ldnull	Push a null reference on the stack.
localloc	Allocate space from the local memory pool.
neg	Negate value.
nop	Do nothing.
not	Bitwise complement.
or	Bitwise OR of two integer values, returns an integer.
pop	Remove the top element of the stack
ret	Return from method, possibly with a value.
shl	Shift an integer left (shifting in zeros), return an integer.
shr	Shift an integer right (shift in sign), return an integer.
stloc	Pop value from stack to local variable
sub	Subtract value2 from value1, returning a new value.
switch	Jump to one of n values.
xor	Bitwise XOR of integer values, returns an integer.

4. Summary

The article presents and analyzes CIL instructions based on prepared code. It shows how high-level language code is mapped to CIL code. It is worth noting the ease with which assemblies compiled in the .NET environment can be examined. CIL code not only can be analyzed, but it can also be easily converted back to high-level code.

5. Bibliography

[1] Price M.J., *C# 14 and .NET 10 – Modern Cross-Platform Development Fundamentals. Build modern websites and services with ASP.NET Core, Blazor, and EF Core using Visual Studio 2026*, p. 18–20, Packt Publishing, 2025.

[2] .NET platform documentation, https://learn.microsoft.com/en-us/dotnet/ (online: 20 January 2026).

[3] IL Spy, https://github.com/icsharpcode/ILSpy (online: 20 January 2026).

[4] *Standard ECMA-335 Common Language Infrastructure* (*CLI*), 6th edition, June 2012, ECMA International.

[5] Coldwind G. (Ed.), *Praktyczna inżynieria wsteczna. Metody, techniki i narzędzia* (*Practical reverse engineering. Methods, techniques, and tools*), pp. 153–166, PWN, 2016.

Język pośredni .NET

M. MOHR

Język pośredni platformy .NET (IL – Intermediate Language, CIL – Common Intermediate Language) to język programowania będący pośrednią wersją między kodem źródłowym a kodem maszynowym. Jest on ustandaryzowany i ściśle powiązany z platformą .NET i językami wysokiego poziomu takimi jak C#, F# i Visual Basic .NET (VB.NET). Platforma .NET wykorzystuje dwuetapowy proces kompilacji. Konwertuje kod źródłowy języka wysokiego poziomu (np. C#) do zestawu platformy .NET, zawierającego kod języka pośredniego, w postaci pliku wykonywalnego (.exe) lub biblioteki łączy dynamicznych (.dll). Dopiero kod pośredni (w czasie wykonywania) ładowany przez środowisko uruchomieniowe .NET (CLR – Common Language Runtime) zostaje przetłumaczony na natywne instrukcje procesora właściwe dla danego systemu operacyjnego i architektury sprzętowej. Dzięki takiemu podejściu ten sam kod CIL może być uruchamiany na różnych platformach takich jak Windows, Linux czy macOS, bez konieczności ponownej kompilacji kodu źródłowego.
Chociaż kod języka pośredniego nie jest pisany bezpośrednio przez programistę, może być analizowany i edytowany przy użyciu specjalnych narzędzi, takich jak ILDASM oraz ILASM, opracowanych przez firmę Microsoft czy niezależnego od wspomnianej korporacji i rozwijanego jako projekt open-source narzędzia ILSpy. Charakterystyczny, dwuetapowy proces kompilacji sprawia, że kod źródłowy skompilowanego i dostarczonego w postaci pliku wykonywalnego programu może być relatywnie łatwo poddany analizie, a nawet odtworzony do postaci zbliżonej do oryginalnego kodu.

Słowa kluczowe: .NET, CIL, język pośredni.