Introduction to IL Assembly Language

(codeproject.com)

Introduction

This article teaches the basics of IL Assembly language which can be used to debug your .NET code (written in any .NET high level language) at low level. From low level, I meant the point where the compiler of your high level language has finished his work. Also, using these basics, you can plan to write your own compiler for a new .NET language.

Table of contents

Whenever you complier your code in .NET, regardless the language you choose, it is converted to Intermediate Language (IL) which is also known as Microsoft Intermediate Language or Common Intermediate Language. You can think the IL as that the Byte Code generated by the Java Language. If you are interested to understand that how .NET deals with data types, and how the code you wrote is converted to IL code etc, then knowledge of IL will give you great advantages. These advantages may include understanding that what the code .NET compiler emits. Hence, if you know the IL, then you can examine the code emitted by the complier and make necessary changes (though not needed in most cases). You can change the IL code to make necessary changes (which your high level language may not allow) to increase the performance of you code. This also may help you to debug your code at low level. And also, if you are planning to write a compiler for .NET, then it is necessary to understand the IL.

IL itself is in the binary format which means it can't be read by the human. But as other binary (executable) code have an assembly language, so the same way IL also has an assembly language known as IL Assembly (ILAsm). IL Assembly has the instruction in the same way as that the native assembly language have. Like, to add two numbers, you have add instruction, to subtract two numbers, you have sub instruction etc. It is obvious that .NET runtime (JIT) can not execute the ILAsm directly. If you have written some code in ILAsm then first you have to compile that to IL code and then JIT will take care of running this IL code.

NOTE: Please note that IL and IL assembly are two different things. Whenever we talk about IL, then it means the binary code emitted by the .NET compiler whereas ILAsm will refer to the IL assembly language which is not in binary form.

NOTE: Please note that I am expecting that you are very much familiar with .NET (including any high level language). In this article, I will not go into details of everything but only those which I think are needed to be explained. If something confuse you, then you can contact me for more discussion.

Introduction to IL Assembly Language

Now let's start the main purpose of this article, the introduction to IL Assembly. ILAsm has the instruction set same as that the native assembly language has. You can write code for ILAsm in any text editor like notepad and then can use the command line compiler (*ILAsm.exe*) provided by the .NET framework to compile that. ILAsm is a tough job for those programmers who have been working in high level languages only but the programmers of C or C++ may adopt it easily. It it's a tough job then we shouldn't waste our time. In IL Assembly, we have to do all the things manually, like pushing values to stack, managing memory etc. Think ILAsm same as that the assembly language but that assembly language deals with native Windows executables and this assembly (ILAsm) deals with .NET executables and also, this assembly is a bit easier and object oriented as well.

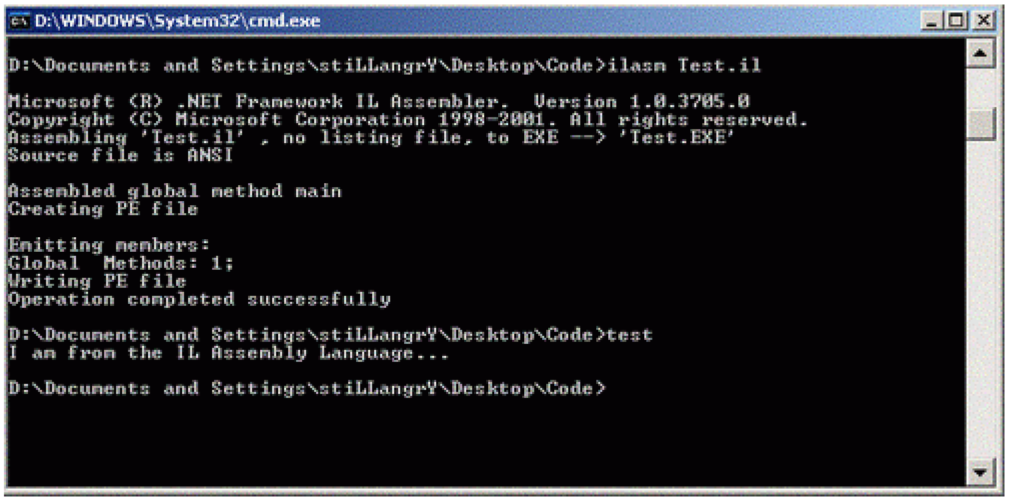
So let's start the beginning of this language with our first example program which will print a single phrase on the screen (console). It is a tradition that the beginning of every language includes a hello world program so we are going to do the same, but the phrase is changed.

```
//Test.IL
//A simple programme which prints a string on the console

.assembly extern mscorlib {}

.assembly Test
{
    .ver 1:0:1:0
}
.module test.exe

.method static void main() cil managed
{
    .maxstack 1
    .entrypoint

    ldstr "I am from the IL Assembly Language..."

    call void [mscorlib]System.Console::WriteLine (string)
    ret
}
```

Figure 1.1 A sample Test program in ILAsm

Write the above code (of fig 1.1) in a simple text editor like notepad and save as *Test.il*. Now let we first compile and run this code and then we will go into details of this. To compile the code, type the following on command prompt

```
ILAsm Test.il  (See the screen shot below)
```



```
D:\Documents and Settings\stiLLangrY\Desktop\Code>ilasm Test.il

Microsoft (R) .NET Framework IL Assembler.  Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
Assembling 'Test.il' , no listing file, to EXE --> 'Test.EXE'
Source file is ANSI

Assembled global method main
Creating PE file

Emitting members:
Global  Methods: 1;
Writing PE file
Operation completed successfully

D:\Documents and Settings\stiLLangrY\Desktop\Code>test
I am from the IL Assembly Language...

D:\Documents and Settings\stiLLangrY\Desktop\Code>
```

Figure 1.2 Output of Sample Test Program. You can see the command I used to compile the code.

ILAsm.exe is a command line tool shipped with the .NET Framework and can be located at \Microsoft.NET\Framework\ folder. You can include this path in your path environment variable. When you have finished compiling your .IL file, then it will output the exe with the same name as that of .IL file. You can specify the output file name using /OutPut= switch like ILAsm Test.il /output=MyFile.exe. To run the outputted exe file, just type the name of the exe and hit return. Output will be before you on the screen. Now let's take some time to understand what we have written in the code. Keep in mind that I am referring to Fig 1.1 while describing the code.

- The first two lines (started with //) are the comments. In ILAsm, you can comment in the same way as that in C# or C++. To comment multiple lines or the part of line, you can also use /* ... */ block as well.
- Next we instructed the ILAsm to import the library named mscorlib (.assembly extern mscorlib {}). In ILasm, every statement started with a period sign (.) indicates that the statement is a special instruction (or directive). So, the .assembly directive here says that we are going to use an external library (that which is not written by us in this code, but pre compiled).
- Next .assembly directive defines the assembly information of our file (.assembly Test ....). In the case of above example, we supplied the "Test" as the name of the assembly and within brackets we supplied some information about the output assembly. That is, the version information. We can provide more information about the assembly in this block, like public key etc.
- Next directive tells the module name of our assembly (.module Test.exe). As we know that there must be at least one module in each assembly.
- Moving to next (.method static void main () cil managed), the .method directive flags that we are going to define a method, which is a static one (same static keyword as in C#) and returns nothing (void). Also, the name of the method is main and it takes no parameter (as there is nothing inside the parenthesis). The final cil managed instructs the compiler to compile this as the managed code.

- Moving inside the method, the first directive is the maxstack (.maxstack 1). This is an important thing which announces the maximum number of items we will load in the memory (evaluation stack actually). We will discuss about this in details a bit later. If you are not aware of this, then skip it for the moment.
- .entrypoint directive tells the compiler to mark this method as the Entry Point of the application, that is, the first function of the program from where the execution will start.
- The next statement (ldstr "I am from the IL Assembly Language...”), ldstr instruction is used to load a string into the memory or evaluation stack. It is necessary to load values into evaluation stack before that can be utilized. We will discuss evaluation stack in details very soon.
- Next statement (`call void [mscorlib]System.Console::WriteLine (string)`) calls (invokes) a method which resides in the mscorlib library. Note that we have given the full signature of that method including the return type, parameter types and also that in which library it resides. We have passed it the string as parameter, which is not a variable, but a data type. The previous statement (`ldstr “I am from the IL…..”`) loaded the string to stack and this method is using the same string to print.
- The final statement, ret, though not needed to explain, instructs to return from the method.

By reading the above lines, you may have an idea that how to write code in ILAsm language. And you might have some idea that ILAsm is not like the high level .NET languages (VB, C#). Anyway, whatever the code you write, you have to follow the structure of this kind (or a little changed when working with classes). Now there are a few things need more discussion. The mainly evaluation stack so, let's do that first.

Evaluation Stack

The evaluation stack can be considered as the normal machine stack, the stack used to store information just before the execution of a statement. We know that the information is stored in the memory when we have to perform some operation on them. Same as we move values to the registers in assembly language before invoking some instruction/interrupt. In the same way we have to move the information (a string in the case of our above example) to the stack before processing (output to screen in above case) that. In the start of our method main (figure 1.1), we notified the runtime of .NET that we are in need to store some information during the course of our method. We said that we will move only one value to stack at one time using `maxstack` directive. Hence if we write our directive as `.maxstack 3,` then runtime will create a room of three values in the stack which we can use at any time. Note that it doesn't mean that we can load only three values in the stack in the life of our method, but it means that we can move maximum three values at one time. Since values are removed from the stack when processing finishes. It should also be noted that whenever the function is called (invoked), the value used in the function are removed from the stack and stack space is emptied. That is how the Garbage Collector works in .NET. Also, there is no limitation that we can move certain type of data to the stack. We can move any kind of data (like string, integer, objects etc) to stack at any time.

Let's take another example which will clear the evaluation stack concept to us.

```
//Add.il
```

```
//Add Two Numbers

.assembly extern mscorlib {}

.assembly Add
{
    .ver 1:0:1:0
}
.module add.exe

.method static void main() cil managed
{
    .maxstack 2
    .entrypoint

    ldstr "The sum of 50 and 30 is = "
    call void [mscorlib]System.Console::Write (string)

    ldc.i4.s 50
    ldc.i4 30
    add
    call void [mscorlib]System.Console::Write (int32)
    ret
}
```

Figure 1.3 *Add.il* Adding two predefined numbers.

The portion above the main method is the same as that of our first example. Only the module name was changed. The thing to discuss is the `.maxstack 2` in the main method which instructs the runtime to allocate enough space in the memory that we can save two values. Then we loaded a string in the stack and printed that out. Next we loaded two integers in the memory simultaneously (using ldc.i4.s and ldc.i4 instructions) and issued the add statement and finally printed an integer type of value. Add statement will look on the stack for two values, if found, it will add them and store the result on the top of the stack. After the add statement, there is another method called Write which writes something on the console. This method requires that there must be some value stored on the top of the stack. In this case, it will look for integer type. If it founds integer value, then it will print that else wise it will raise an error.

Do not confuse with ldc.i4.s and ldc.i4, both represents to integer data type, but first one is of single byte and second of four byte.

I hope that you have understood the way of using evaluation stack and also that how it works. Now let's move to discuss more things about the ILAsm language.

IL Data Types

As to learn any language, we first discuss the data types used in that language. So the same case is here. Let's take a look on the table below (figure 1.4) to know about the data types of IL Assembly. But before moving to that, I would like to point to one thing that there is no consistency in .NET data type definition in different languages. Like an integer (32 bit) in VB .NET is defined using Integer but in C# and VC++, it is int; though in both cases it is representing System.Int32. And also, we need to remember that is it Common Language Specification (CLS) Compliant or not. Like the UInt32 which is not recognized by VB .NET and also it is not CLS Compliant.

Well, let's start to remember the table which provides new names of the data types for ILAsm Language.

IL Name	.NET Base Type	Meaning	CLS Compliant
Void		no data, only used as return type	No
Bool	System.Boolean	Boolean Value	No
Char	System.Char	Character Value (16 bit unicode)	No
int8	System.SByte	Single Byte Integer (signed)	No
int16	System.Int16	Two Byte Integer(signed)	No
int32	System.Int32	Four Byte Integer(signed)	Yes
int64	System.64	8 Byte Integer(signed)	Yes
native int	System.IntPtr	Signed Integer	Yes
unsigned int8	System.Byte	One byte integer (unsigned)	No
unsigned int16	System.UInt16	Two byte integer (unsigned)	No
unsigned int32	System.UInt32	Four byte integer (unsigned)	No
unsigned int64	System.UInt64	Eight byte integer (unsigned)	Yes
native unsigned int	System.UIntPtr	Unsigned Integer	Yes
Float32	System.Single	Four byte Floating Point	No
Float64	System.Double	Eight byte Floating Point	No
object	System.Object	Object type value	Yes
&		Managed Pointer	Yes
*	System.IntPtr	Unmanaged Pointer	Yes
typedef	System.Typed Reference	Special type that holds data and explicitly indicates the type of data.	Yes
Array	System.Array	Array	Yes
string	System.String	String type	Yes

Figure 1.4 Datatypes in ILAsm

We also have some mnemonics of the data types in ILAsm like .i4, .i4.s, .u4 etc. as we used in the above example. Types listed above are those recognized by the ILAsm and also the table mentions that which types are CLS Compliant and which not. So, keeping in mind the above types, we can call any function like this

```
call int32 SomeFunction (string, int32, float64<code lang=msil>)
```

Which means, the function `SomeFunction` returns a value of type `int32` (`System.Int32`), and takes three values of type `string` (`System.String`), `int32` (`System.Int32`) and `float64` (`System.Double`) respectively. Note that these were the basic data types of CLR and ILAsm. If we are interested to deal with not basic data types (user defined) then we may deal like this.

```
//In C#
ColorTranslator.FromHtml(htmlColor)
//In ILAsm
call instance string [System.Drawing]
    System.Drawing.ColorTranslator::ToHtml(valuetype
    [System.Drawing]System.Drawing.Color)
```

Note that we explicitly defined the parameter types. We also defined the namespace where that type reside and a keyword value-type which flags that we are about to reference any non basic data type.

The things will be clearer in the coming section when we will write a sample program which will deal with the types. But first, let's take a look at the basics of the language like variable declaration, loops, conditions etc.

Variable Declaration

Variables are the major part of any programming language and hence, ILAsm also provides us a way to declare and use variables. Though not so simple as that in higher languages (VB .NET, C#) .locals directive can be used to declare the variables. This directive should usually occur on the beginning of any method though you can put your declaration at any place, but obviously, before using them. Here is a sample which can declare the variable, sets the values and then use them to print.

```
.locals init (int32, string)
ldc.i4 34
stloc.0
ldstr "Some Text for Local Variable"
stloc.1
ldloc.0
call void [mscorlib]System.Console::WriteLine(int32)
ldloc.1
call void [mscorlib]System.Console::WriteLine(string)
```

Figure 1.5 Local variables

We declared two variables using .locals directive, one of type `int32` and other of `string`. Then we loaded a value 34 of type `int32` to memory and assign that to local variable zero, which is first variable actually. Note that in ILAsm variables variables can be accessed by their index (number of definition) and these numbers starts with zero. Then we loaded a string into memory and assigned that to second variable. And finally, we printed both variables. ldloc.? can be used to load any type of variable value to memory (either integer, double float or object).

I didn't use variable names here. Since those are local and we are not intended to expose them out of the method. But it doesn't mean that you can not declare variables by name. Sure, you can. To declare local variables you can name the variables with their data types, just like in C# like .locals init (int32 x, int32 y)

And later, you can load or set the values of these variables using the same statements, but with variable names like `stloc x` and `ldloc y`. Though you have declared your variables using names but you can still access them with their numbers like ldloc.0, stloc.0 etc Note: throughout the code in this article, I am using the variables without names.

Now you have the idea to handle the variables and stacks. Please review the codes provided above if you are facing any problem because after now, we will have some tough task while playing with the stack. We will frequently move data to memory and get back. So a good understanding of initializing variables, setting values to variables and loading values to stack from variables is necessary.

Decisions / Conditions

Decisions or conditions are other necessary objects of any programming language. In the low level languages, like native Assembly language, the decisions are made using jumps (or branch). So the same is with the ILAsm. Take a look at the code snippet below.

```
br JumpOver         //Or use the br.s instead of br
//Other code here which will be skipped after getting br statement.
//
JumpOver:
//The statements here will be executed
```

Compare this statement with the `goto` statement written in any high level language which transfers the control to some label written after the `goto` statement. But here, br is used instead of `goto`. br.s can also be used if you are sure that the target is within the -128 to +127 byte of the br statement since it will use int8 instead of int32 for branching offset. The above method was unconditional branching since there was no condition evaluated before the br statement and the code will always jump/branch to the JumpOver label. Let's see a code snippet which can describe the use of conditional branching means, the jumping (or branching) through some logical test.

```
 //Branching.il
.method static void main() cil managed
{
    .maxstack 2
    .entrypoint
    //Takes First values from the User
    ldstr "Enter First Number"
    call void [mscorlib]System.Console::WriteLine (string)

    call  string [mscorlib]System.Console::ReadLine ()
    call int32 [mscorlib]System.Int32::Parse(string)

    //Takes Second values from the User
    ldstr "Enter Second Number"
    call void [mscorlib]System.Console::WriteLine (string)

    call  string [mscorlib]System.Console::ReadLine ()
    call int32 [mscorlib]System.Int32::Parse(string)

    ble Smaller
        ldstr "Second Number is smaller than first."
        call void [mscorlib]System.Console::WriteLine (string)
    br Exit

Smaller:
    ldstr "First number is smaller than second."
    call void [mscorlib]System.Console::WriteLine (string)
Exit:
    ret
}
```

Figure 1.6 *Branching.il* (only main method, included)

The above program takes two values from the user and then check for the smaller number. The statement which needs attention is the “ble Smaller” which instructs the compiler to check if the first value in the stack is less than or equal to second, then it should branch to Smaller label. And if it is not then no branching will take place and the next statement will be executed which is to load a string and then to print that out. After this, an unconditional branching is occurred which was necessary because if that is not here then according to the flow of program, the statement after the Smaller label will be executed. So, we issued “br Exit” which caused the program to branch to Exit label and executed the ret statement.

Other conditions includes `beq` (==), `bne`(!=),`bge` (>=),`bgt`(>), `ble` (<=), `blt`(<) and also `brfalse` (if top item in the stack is zero) and `brtrue` (if top item in the stack is not zero). You can use any of then to execute some part of your code and skip other. As I mentioned earlier, there is no facility in ILAsm as we have in high level languages. So everything should be done yourself if you are still planning to write some code in ILAsm.

Loops

Another part of the basics of language is the Loops. Loop is nothing but the repetition of the same block of code again and again. It involves the branching actually depending on the value of a variable called loop index. Again, you have to look at a code and have to spend a little time to understand that how the loops work.

```
 .method static void main() cil managed
{
    //Define two local
    variables .locals init (int32, int32)
    .maxstack 2
    .entrypoint
    ldc.i4 4

    stloc.0        //Upper    limit of the Loop, total 5
    ldc.i4 0
    stloc.1        //Initialize the Starting of loop

Start:
    //Check if the Counter exceeds
    ldloc.1
    ldloc.0
    bgt Exit //If Second variable exceeds the first variable, then exit

    ldloc.1
    call void [mscorlib]System.Console::WriteLine(int32)

    //Increase the Counter
    ldc.i4 1
    ldloc.1
    add
    stloc.1
    br Start
Exit:
    ret
}
```

Figure 1.7 *Loops.il* (only main method)

While the same code may written in higher languages, like C#, should look like this

```
for (temp=0; temp <5; temp++)
            System.Console.WriteLine (temp)
```

Let's examine the code. First of all, we declared two local variables and initialized the first variable with value of 4 and also the second variable with zero. The real loop starts from the Start Label, from where first we checked either the loop counter (variable 2, ldloc 1) exceeds the upper limit of loop (variable 1, ldloc 0), if that is the case then program will jump to Exit label which will cause to terminate the program. If this is not the case, then the value will be printed on the screen and one increment will be made in the variable and code will jump to Start label again to perform check if counter exceeds the upper limit or not. This is the way the loops works in ILAsm.

Defining Methods

We have seen about the decisions (conditions or branching), Loops and also declaring variables. Now is the time to see that how methods can be created in the ILAsm. The method of declaring methods in ILAsm is almost the same, and I hope that you have guessed till now, as that in C# or C++ but with a little changes. So, let's look at the code snippet first then we will discuss what we did.

```
//Methods.il
//Creating Methods

.assembly extern mscorlib {}

.assembly Methods
{
    .ver 1:0:1:0
}
.module Methods.exe

.method static void main() cil managed
{
    .maxstack 2
    .entrypoint

    ldc.i4 10
    ldc.i4 20
    call int32 DoSum(int32, int32)
    call void PrintSum(int32)
    ret
}
```

```
.method public static int32 DoSum (int32 , int32 ) cil managed
{
    .maxstack 2

    ldarg.0
    ldarg.1
    add

    ret
}
.method public static void PrintSum(int32) cil managed
{
    .maxstack 2
    ldstr "The Result is : "
    call void [mscorlib]System.Console::Write(string)

    ldarg.0
    call void [mscorlib]System.Console::Write(int32)

    ret
}
```

Figure 1.7 *Methods.il* Define and call your own methods

A simple program which adds two numbers (pre defined, for the sake of simple code) and print the result. We defined two methods here. Note that both methods are static so that we can directly use them without creating any instance. First we loaded two numbers on the stack and called the first method `DoSum` which was expecting two int32 values on the stack. Coming to the body of the function, as the declaration looks the same as that of main and we have seen it many times till now, we again defined the `maxstack` but note that we didn't included the `.entrypoint` directive here since one program can have only one entry point and in the case of above example, we have declared main method as entry point. The ldarg.0 and ldarg.1 instructions make the runtime to load arguments on the evaluation stack, the argument passed to the method. Then we simply added them using add statement and then method returns. Note that the method returns an int32 type of value. Now which value should it return? Of course, the value which was available on the stack when add statement finished its work. From here control will be transferred back to main method from where it will call another method `PrintSum`. `PrintSum` also requires one int32 type value. Now it should be noted that `DoSum` method returned int32 type value, and that is in the evaluation stack; meanwhile we called `PrintSum` method which was also looking for an int32 type value. Hence, the returned value from the `DoSum` method will be passed to `PrintSum` and in `PrintSum`, it first prints a simple string on the screen, then loads the argument using ldarg.0 and then print that too.

The above way says that creating method is not a hard task in ILAsm. Yes, actually it is. But methods do get values by reference. So let's take a look on that too.

Passing Variables by Reference

IL also supports the by reference values and of course it should because high level languages in .NET supports by reference arguments and the code from high level languages is converted to IL code and we are discussing IL Assembly language which produces the same IL code. Whenever we pass any variable by reference, then the address of the memory location, where that variable value was stored, is passed in contrast to the by value method, in which the copy of the value is passed to the function. Let's see an example of how by reference approach works in IL Assembly.

```
.method static void main() cil managed
{
    .maxstack 2
    .entrypoint
    .locals init (int32, int32)

    ldc.i4 10
    stloc.0
    ldc.i4 20
    stloc.1

    ldloca 0          //Loads the address of first local variable
    ldloc.1 //Loads the value of Second Local Variable
    call void DoSum(int32 &, int32 )
    ldloc.0
    //Load First variable again, but value this time, not address
    call void [mscorlib]System.Console::WriteLine(int32)
    ret
}
.method    public static void DoSum (int32 &, int32 ) cil managed
{
    .maxstack 2
    .locals    init (int32)
    //Resolve the address,    and copy value to local variable
    ldarg.0
    ldind.i4
    stloc.0
    ldloc.0
    //Perform the Sum
    ldarg.1
    add
    stloc.0

    ldarg.0
    ldloc.0
    stind.i4

    ret
}
```

Figure 1.8 *MethodRef.il* Passing variables by reference.

The interesting thing in the above example is the use of some new instructions like `ldloca`, which loads the address of variable, instead of the value in the stack. In the main method, we declared two variables (local) and assigned them some values (10 and 20 respectively). Then we loaded first variable's address to the memory and value of the second variable and invoked the `DoSum` method. If you see the `DoSome` method signature (calling), then you will notice one thing that we used & with the first int32 (parameter) which mentions that the stack contains memory reference instead of the value and we are interested to pass variable by reference. In the same way, `DoSome` method declaration also contains the same & with the first parameter it receives which also, sure not to mention, says that the variable will be passed by reference. So, one variable is passed by reference and second by value (normal). Now the problem was to resolve the address to the value so that we can perform some action on the value and then set to the variable if necessary. For this reason, we loaded first argument to the stack (which contains the address of actually variable passed) and invoked ldind.i4 statement which loads the value of an integer (32 bit) by taking address from the stack (goes to that address, reads value and put on the stack). We stored that value in a local variable so that we can reuse that easily (or we have to perform these steps again and again). Then, simply, we loaded that local variable and second argument (which was by value) to the stack and added them and stored in the same local variable. Now there is one more interesting thing here that we changed the value on the memory location which was representing the first argument's value (the argument passed by reference). We did that by first loading argument 0 (by reference argument) to the stack which will actually load the address of original variable passed to this method, then the value we want to set, and finally the statement, stind.i4 which is just opposite to the ldind.i4 statement we used above. It set's the value on the memory location which is available on the the stack. To test the changed value, we simply printed that in the main method. Note that the `DoSum` method do not return anything, and in the main method, we simply loaded first local variable again (now the value, not the memory reference) and used `WriteLine` method to print that.

This was the way the ILAsm deals with the by reference variables. Up to here, we have seen the methods of playing with variable declaration, conditions, loops and methods (by value parameters and by reference parameters). Now is the time to declare our namespaces and classes using ILAsm language.

Creating Namespaces and Classes

Yes, and of course, it is possible in ILAsm to create your classes and namespaces. Actually, it is fairly easy to create your class or namespace in ILAsm as that in any higher level language. Don't believe? Then let's see.

```
//Classes.il
//Creating
Classes
.assembly extern

mscorlib {} .assembly Classes

{ .ver 1:0:1:0    }
.module Classes.exe
.namespace HangamaHouse

{
    .class public ansi auto Myclass extends [mscorlib]System.Object
    {
        .method public static void main() cil managed
        {
            .maxstack 1
            .entrypoint

            ldstr "Hello World From HangamaHouse.MyClass::main()"
            call void [mscorlib]System.Console::WriteLine(string)

            ret

        }
    }
}
```

Figure 1.9 *Classes.il* Creating your own namespace and class.

I think, now it is not much needed to explain the code. The thing is very simple, .namespace directive, followed by a name HangamaHouse, was issued to tell the compiler that we are going to create a namespace of name HangamaHouse. Within the block of namespace, we introduced a new class with .class directive and mentioned that this class is public and it extends (inherits) from System.Object class. This class contains only one method which is static and public. Rest the code of the method you knows very well.

Here I would like to mention that all classes you create are inherited from the Object class if you do not mention any inheritance. Like in this case, we explicitly mentioned that our class is inheriting the Object class of System namespace, If we do not mention it here, then it will still inherit the Object class. Yes, there is a case that our class inherits from any other class, then it will not inherit from Object class (but that class from which you are inheriting your class may inherit the Object Class).

There were two more keywords used in the above class creation, those are ansi and auto. Ansi tells that all the strings in the class should be converted to ANSI strings. Other options for this are unicode and autochar (conversion will be determined automatically according to platform). Second one is auto keyword which specifies that the runtime will automatically choose appropriate layout for members of the object in unmanaged memory. Other options for this are sequential (members are laid out sequentially) and explicit (layout is explicitly defined). For more information, see StructLayout or LayoutKind enumeration in MSDN. auto and ansi are the default keywords for the class and if you do not define anything here, these will be assumed automatically.

Scope of the Objects (Member Accessibility Flags)

The following table summarizes the scope of the Classes in ILAsm.

ILAsm Name	Description	C# Name
Public	visible to class, namespace and objects (all)	public
Private	visible inside the class only	private
Family	visible to class and derived classes only	protected
assembly	visible within same assembly only	internal
familyandassem	visible within derived classes of the same assembly	N/A
familyorassem	visible to derived classes and those of the same assembly	protected internal
privatescope	as that the private, but it can not be refereneced	N/A

Figure 1.10 The Member accessibility flags for ILAsm

There are some more scopes which can be used with the methods and fields (variables in the class). You can find out the complete list in the MSDN.

Creating and Using Class Objects

In this section of my article, I will show you that how can you create the instance of your class and use in ILAsm code. Before this, you have seen that how to create your own namespaces and classes in ILAsm. But creating something is useless until we can't use that. So, let's begin to create a simple class and use that.

Let's create our own library in ILAsm. This simple library contains only one public method. That is, it will receive one value and will return the square of that value. Simple is the best to understand. Look at the code.

```
.assembly extern mscorlib {}
.assembly MathLib
{
    .ver 1:0:1:0
}

.module MathLib.dll

.namespace HangamaHouse
{
    .class public ansi auto MathClass extends [mscorlib]System.Object
    {

        .method public int32 GetSquare(int32) cil managed
        {
            .maxstack 3
            ldarg.0        //Load the Object's 'this' pointer on the stack
            ldarg.1
            ldarg.1
            mul
            ret

        }
    }
}
```

Figure 1.11 the *MathLib.il* Library for math operation Square

Note: Compile above code to a DLL file. Use ILAsm MathLib.il /dll

The code looks simple. It defines one namespace with the name HangamaHouse and one class inside that namespace, named MathClass, which extends from the Object class of System namespace as our above class (in figure 1.10) do. Now within that class, we defined one method named GetSquare which takes one argument of type int32. We defined the `maxstack` to 3 and then loaded the argument zero. Then we again loaded argument one two times. Wait a minute, we are receiving only one argument here but we have loaded argument zero and one (two arguments total). How is that possible? Well it is. Actually, the zero argument (ldarg.0) is the Object's reference to "this" pointer. Since every instance object is always passed the address of the object's memory. So our actual argument(s) starts from index 1. Well, we loaded argument 1 two times so that we can multiply them which we did with mul instruction. The result of the multiplication was stored on the stack which was returned to the calling method as we issued the ret instruction immediately.

The library building was no matter. Simple yet, now let's look at the example which uses this library

```
.assembly extern mscorlib {}
```

```
.assembly extern MathLib {.ver 1:0:1:0}
//
//rest code here
//
.method static void    Main() cil managed
{
    .maxstack 2
    .entrypoint
    .locals init (valuetype [MathLib]HangamaHouse.MathClass mclass)

    ldloca mclass
    ldc.i4 5
    call instance int32 [MathLib]HangamaHouse.MathClass::GetSquare(int32)
    ldstr "The Square of 5 Returned : "
    call void [mscorlib]System.Console::Write(string)
    call void [mscorlib]System.Console::WriteLine(int32)

    ret
}
```

Figure 1.12 Using the MathLib library's Class MathClass

First two lines of this method are simple. Come to third line where we defined a local variable of type MathClass (From HangamaHouse namespace. Note that we have imported this library (MathLib) with the mscorlib library. We also provided the version number which is, though, not necessary as we have been using external library mscorlib since from the start of this article without providing the version. Again note that we wrote valuetype before the type of this object we are going to create and provided the complete signature of the class including library name. Next we loaded the address of the local variable mclass. Then loaded an integer with value 5 and called the method GetSquare from MathClass object mclass. This was done since we have loaded the mlcass object just a couple of statements ago. The memory reference of the mclass object was available on the stack. When we called the MathClass's GetSquare method, then it looked the stack for the object's reference and the values for the parameter, if found them and called the method using that reference. One another thing you should note that we used instance keyword while calling the method which we have never used. Instance keyword tells the compiler that we are going to call the method of any object's instance; it is not the static method. After completing the execution, GetSquare method returned a value of type int32 which was stored on the stack and we used that with a string to print on the console.

So the main thing here was to declare the object of the class which was done using .locals directive and valuetype and complete signature of the class including library. Secondly calling method from the class, which was done by first loading the class object into the memory, then loading any value which will be passed to the method and then finally used instance keyword while calling method.

In the same way, we can use properties and constructors in the class and can call them in our code. Next section of my article describes that how can you create your private fields, constructors and properties in ILAsm language and how can you call them using the same ILAsm code.

Creating Constructors and Properties

Constructor is actually a method which is called in high level languages whenever the object of the some class is created. But in low level languages like ILAsm, you manually have to call them though there is no doubt that it is a methods which always returns nothing. The sample code below demonstrates how to create constructors. I am including only the necessary part here, the source code with this article includes complete code for this section. Please be attentive while reading this section because this section will teach you a lot of things.

```
.class public MathClass extends [mscorlib]System.ValueType
{
    .field private int32 mValue
    //Other code goes here

    .method public specialname rtspecialname
        instance void .ctor() cil managed
    {

        ldarg.0
        ldc.i4.s    15
        stfld int32 HangamaHouse.MathClass::mValue
        ret
    }
    //Other code goes here.
```

Figure 1.13 The constructor for the `MathLib` Class

First thing you may notice here is that I used to inherit my class from `System.ValyeType` not from Object. Well, since in .NET a class is actually a type so it is always inherited from the `ValueType`. Though you might have been inheriting your classes from other classes, but if you look at the inheritance tree, then you will end up with `ValueType`. Yes, I did inherited my classes from Object class before this, but that was not the right time to describe this and those were not complete classes as we are going to create a complete class here (with constructor, properties etc). If you do not want to take advantages of these full features of the class, then you can inherit your class from anywhere.

After the declaration of the class, I declared a private field named mValue (private variable in high level languages). Then I declared a constructor using .method directive. Remember, constructor is a method. Now you will surprise that the .ctor is used instead the name of the class (as in high level languages like C++). Yes, the .ctor represents the constructor method in ILAsm. This is the default constructor since it takes no parameter. What we did there is, loaded the reference of object itself (this) using ldarg.0 statement, then we loaded an integer of value 15 and assigned that value to our private field mValue. stfld statement can be used to set the value of any field. We provided it the complete signature of the field. I think you should not surprise now that why we did so. Finally, we returned from the method (constructor).

You should also notice that we used a couple of more keywords in the declaration of this constructor. Those are `specialname` and `rtspecialname`. Actually, these tell the language runtime to treat this method name as a special name. You can use them with the constructors and properties etc. But these are not necessary.

Unlike the high level languages, in ILAsm constructor is not called automatically. You explicitly have to call it. Here is a code snippet which calls the constructor to initialize the value of the class.

```
.method public static void Main() cil managed
{
    .maxstack 2
    .entrypoint
    .locals init (valuetype [MathLib]HangamaHouse.MathClass mclass)

    ldloca mclass
    call instance void [MathLib]HangamaHouse.MathClass::.ctor()
```

The above code creates a local variable named mclass of type `MathClass` which resides in the HangamaHouse namespace. Then it loads the address of that to stack and then calls the constructor (.ctor method). If you examine then it is the same way as we call normal methods in the ILAsm. No difference. Similarly, you can define the overloaded constructor while creating a new .ctor method which accepts the parameters and can call in the same way, as we called this one.

As about the properties, so the properties are also the methods actually. See the figure and then we will be able to understand that fully.

```
.method  specialname public instance int32 get_Value() cil managed
{
    ldarg.0
    ldfld int32 HangamaHouse.MathClass::mValue
    ret
}
.method specialname public instance void set_Value(int32 ) cil managed
{
    ldarg.0
    ldarg.1
    stfld int32 HangamaHouse.MathClass::mValue

    ret
}
//Define the property, Value
.property int32 Value()
{
    .get instance int32 get_Value()
    .set instance void set_Value(int32 )
}
```

Figure 1.15 Properties

You can examine the code above and can surely say that it is the same as that of methods code. But you can see one another thing here; that is .property directive. It defines two things inside its body. That is, the property get and property set. This actually marks the methods as the part of the property and put them under one tree. Since we can see that the both methods are defined above, the `get_Value` and `set_Value` methods. Those are simple one as we have been looking in this article. Calling any property is also fairly simple as we have seen that these works like the methods which actually are.

```
.maxstack 2 .locals
init (valuetype [MathLib]HangamaHouse.MathClass tclass)

ldloca tclass
ldc.i4 25
call instance void [MathLib]HangamaHouse.MathClass::set_Value(int32)
ldloca tclass
call instance int32 [MathLib]HangamaHouse.MathClass::get_Value()
ldstr "Propert Value Set to : "
call void [mscorlib]System.Console::Write(string)
call void [mscorlib]System.Console::WriteLine(int32)
```

Figure 1.16 Using Properties. Also a method is called in this code `GetSquare`

Not surprising. We created the instance of the class and then called the `set_Value` method (which is actually a property and we are going to set the value of the property). Then for confirmation, we re-read the value of that property and printed that with a string.

Up to here, we have covered a lot of things which may help you to start working in ILAsm language. But there is one necessary thing left, the errors and debugging.

Creating Windows Form (Frame)

This section demonstrate that how can you combine the information provided above to create a simple GUI, the Windows Form. In this application, I have created a simple form from `System.Windows.Forms.Form` class. It do not contain any control but I have changed the properties of the form like `BackColor`, `Text` (title) and `WindowState`. The code is very easy and step to step. So, let's take a look at this final code before closing my article.

```
.namespace MyForm
{
  .class public TestForm extends
       [System.Windows.Forms]System.Windows.Forms.Form
  {
    .field private class [System]System.ComponentModel.IContainer components
    .method public static void Main() cil managed
    {
      .entrypoint
      .maxstack  1

      //Create New Object of TestForm Class and Call the Constructor
      newobj     instance void MyForm.TestForm::.ctor()
      call       void [System.Windows.Forms]
          System.Windows.Forms.Application::Run(
          class [System.Windows.Forms]System.Windows.Forms.Form)
      ret
    }
```

Figure 1.17 Windows Form's EntryPoint Method

This is the entry point of our application. First of all (after creating class `TestForm` in the namespace `MyForm`) we defined a local variable (field) of type `IContainer`. Note that we mentioned class before defining the type of field. Then in the Main method, we created the object of TestForm using newobj instruction. Then we called the Application.Run method to start the application. If you compare it with high level language code, then you will notice that it is the same technique as used there. Now let's see the .ctor method of our class (TestForm).

```
.method public specialname rtspecialname instance
    void  .ctor() cil managed
{
  .maxstack  4

  ldarg.0
 //Call Base Class Constructor
  call       instance void [System.Windows.Forms]
                  System.Windows.Forms.Form::.ctor()

  //Initialize the Components
  ldarg.0
  newobj   instance void [System]System.ComponentModel.Container::.ctor()
  stfld    class [System]System.ComponentModel.IContainer
                  MyForm.TestForm::components

  //Set the Title of the Window (Form)
  ldarg.0
  ldstr      "This is the Title of the Form...."
  call   instance void [System.Windows.Forms]
                 System.Windows.Forms.Control::set_Text(string)

  //Set the Back Color of the Form
  ldarg.0
  ldc.i4 0xff
  ldc.i4 0
  ldc.i4 0
  call valuetype [System.Drawing]System.Drawing.Color
               [System.Drawing]System.Drawing.Color::FromArgb(
               int32, int32, int32)

  call   instance void [System.Windows.Forms]
               System.Windows.Forms.Control::set_BackColor(
               valuetype [System.Drawing]System.Drawing.Color)


  //Maximize the Form using WindowState Property
  ldarg.0
  ldc.i4 2        //2 for Maximize
  call instance void [System.Windows.Forms]
               System.Windows.Forms.Form::set_WindowState(
               valuetype [System.Windows.Forms]
               System.Windows.Forms.FormWindowState)
  ret
}
```

Figure 1.18 .ctor method (Constructor) of TestForm

Very simple, we simply called the base class's .ctor method (constructor). Then we created the object of Container and assigned that to out component object (field of our class). Form initialization completed here. Now we are moving to set some properties of our new form. First of all we are going to set the title of our form (Text property). We loaded a string to stack and called the set_Text method of Control object (since Text property is inherited from the control). After setting the Text property we initiated our work to set BackColor property. We called the FromArgb method to get color from Reg, Green and Blue value. We first loaded these three values to stack and then called the Color.FromArgb method to get an object of Color class so that we can assigned that to Form's BackColor property. We assigned that to form's BackColor property in the same way as we have been setting properties before this. Finally, we set the WindowState property of form to Maximized. Same thing, same way. Note that we loaded an integer value to the stack and assigned that to FormWindowState enumeration, since Enums has actually the variables with predefined values.

Although, the code for creating a Windows Form finishes here, but we can also define the Dispose event (Destructor of the form) so that we can clean the memory by releasing unnecessary objects. If you are interested to know about the code of Dispose event, then just look below to this line.

```
    .method family virtual instance void Dispose(bool disposing) cil managed
    {
      .maxstack  2

      ldarg.0
      ldfld      class [System]System.ComponentModel.IContainer
                   MyForm.TestForm::components
      callvirt   instance void [mscorlib]System.IDisposable::Dispose()

        //Call Base Class's Dispose Event
      ldarg.0
      ldarg.1
      call       instance void [System.Windows.Forms]
                    System.Windows.Forms.Form::Dispose(bool)
      ret
    }
```

The Dispose method is overloaded, so it is declared as virtual. We simply loaded the object's reference (this), loaded the components field, and called the Dispose method of IDisposable. Then we called the Dispose method of our form. That's all.

So, creating the user interface was not very hard task (though a little). From here, you can code to add components such as textboxes, labels etc in your form and also can code at their events. Can you?

Errors and Debugging

Errors are the part of any programming language. So ILAsm is not excluded. Like errors of other programming languages, in ILAsm you may also encounter with Compiler errors (normally known as syntax error), runtime errors and logical errors. I am not going in details that what these errors are since all of you know them very well. The purpose of this section is to introduce some tools and techniques which may help you to debug your program. First of all, you can generate debug info file while compiling your code. Just use the /debug switch with *ILAsm.exe* while compiling your code like

```
ILAsm.exe Test.il /debug
```

This will produce one exe file of name *Text.exe* and also one debug information file *Test.pdb*. Now this file can be used to later debug your application.

You can use a tool to verify your application (an assembly actually). That is PE Verify (*peverify.exe*), shipped with .NET Framework SDK and you can locate that at *C:\Program Files\Microsoft .NET\Framework SDK\Bin* folder (by default). Pre Verify tool do not use source code to verify the assembly, but it actually takes an exe to examine if the compiler has emitted the invalid code. There may be some case when you need to verify your assembly like you may be using some third party compiler or writing your own. Usage is very simple. Consider the example below.

```
peverify.exe Test.exe
```

For more information and options, you can see command line switches of *peverify.exe* using *peverify.exe /?*

You can also get the IL code from any of your .pre-compiled EXE or DLL (in .NET) using *ILDasm.exe. ILDasm.exe* is another valuable tool shipped with .NET Framework SDK which can help to debug your applications at low level. This tool is useful to you if you have written your code in any high level .NET language and want to see what IL code was generated by your compiler. It can be located under the same folder as that *peverify.exe* (Framework SDK\Bin folder). You can get IL code of any of your .NET EXE file using this way.

```
ILDasm.exe SomeProject.exe /out:SomeProject.il
```

There are a number of other tools which can be used to debug your .NET applications like *DbgClr.exe, CorDbg.exe.* You can find their reference from MSDN, .NET Framework SDK or can search the web for third party.

Summary

In this article we learned the basics of ILAsm and wrote some programs using ILAsm language. We started from the basics of ILAsm and that why it is needed. Then we wrote a sample test program which prints a single line on the console. We learned a little about the Evaluation stack and also saw via a simple code (adding numbers) that how it works. Then we gathered knowledge about IL data types and used them to declare variables, applying conditions, creating loops etc. We also defined methods and after that we switched to the namespace and class creation. There we created the objects of our own classes and used them. Also we created the constructors and properties for our classes.

Conclusion

Writing code in ILAsm is not a simple task. As there were a number of things not discussed in this article like arrays, error handling etc. But after getting familiar with ILAsm, you can proceed to work on them yourself. ILAsm can be helpful to you if you want to debug your code at low level or if you are planning to write some compiler for .NET platform. If you are just a beginner in .NET then I will not recommend you to go through this process since it needs a good understanding of .NET, but it will give you better understanding that how .NET Common Language Runtime is working behind the scene.

About the Author

Sameers (theAngrycodeR) has did his Master in Computer Sciences in Feb. 2002. He is the Project Manager in City Soft (Pakistan) and has developed and managed a number of projects. He proved his expertise in Visual Basic 6 and now working in VB .NET. He is the author of many articles on CodeProject.com as well as on the Microsoft .NET Community Site, GotDotNet. Also submitted many source codes on Planet-Source-Code.com. The complete information about the author can be found here.

License

About the Author

Sameers (theAngrycodeR)

Occupation:	Web Developer
Location:	United States