Lcc.NET: Targeting the .NET Common Intermediate Language from Standard C

David R. Hanson
Microsoft Research
drh@microsoft.com

November 2002, Revised April 2003, June 2003

Technical Report MSR-TR-2002-112

Abstract

The core of the Microsoft .NET platform includes a new virtual machine, the Common Intermediate Language, also known as MSIL. Unlike most other VMs, including the Java VM, MSIL is designed specifically to support a wide range of languages. While it is designed primarily for type-safe, object-oriented languages, it also has facilities that support both low-level languages and very high-level languages. For example, it accommodates unsafe pointer arithmetic and tail calls. This paper describes the implementation of an MSIL back end for lcc, a retargetable compiler for Standard C. C is at one end of the range of languages that MSIL intends to support, and lcc is about the simplest "real" C compiler widely available. Porting lcc to MSIL thus provides a realistic test of how well MSIL supports this class of languages and provides a glimpse of its performance costs. This effort succeeded, but static initializations, function pointers, separate compilation, and address arithmetic were major problem areas. These problems also suggested improvements in lcc's code-generation interface, and they exposed a long-standing error in the lcc front end. Programs compiled by the MSIL back end run 2-3 times slower than those compiled by lcc native Intel x86 back end, but the MSIL programs have some important diagnostic benefits.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com/

Lcc.NET: Targeting the .NET Common Intermediate Language from Standard C

Introduction

The Microsoft .NET platform includes tools, technologies, and methodologies for writing internet applications [13]. It includes new programming languages, tools that support XML web services, and new infrastructure for writing HTML pages and Windows applications. Its core components are a new virtual machine and runtime environment.

The virtual machine, called the .NET Common Intermediate Language or MSIL for short, provides a low-level, executable, type-safe program representation that can be verified before execution, much in the same way as the Java VM [11] provides a verifiable representation for Java programs. Unlike the Java VM, however, MSIL is designed to support multiple languages, from type-unsafe imperative languages to very high-level functional languages. Its multi-language design ambitions are close to those of UNCOL [14], but its target machines are limited to modern processors with byte-addressing, two's-complement integer arithmetic, and IEEE 754 floating-point arithmetic. MSIL and the runtime environment are nearing acceptance as an ECMA standard [1].

While MSIL is intended to support nearly any language, the Microsoft languages Visual Basic .NET, C# [7], and Managed C++, a type-safe version of C++ similar in spirit to the design described in Reference 2, are the only supported languages in wide use at the present time.

Language implementors are often eager to target their favorite language to a new VM, particularly one that is likely to be used widely. There are many efforts underway to target MSIL for a wide variety of languages, from COBOL to Haskell. Component Pascal [6] is perhaps the best-documented effort to date.

This paper describes the design and implementation of an MSIL back end for lcc [4], a retargetable compiler for Standard C. MSIL is class-based; that is, it supports classes and packages functions only in classes. Consequently, it is best suited for object-oriented languages. It does, however, have usual set of imperative instructions abstracted from modern processors, which makes it possible to handle any language, at least in theory. Interestingly, C's lack of classes and objects, its unsafe features, such as pointer arithmetic, and its use of separate compilation, put it at one end of the languages spectrum for MSIL.

Lcc is about the simplest "real" C compiler available, and it is well documented. Targeting MSIL thus helps pinpoint the low end of this language spectrum and identifies some of the implementation problems for similar languages. As with all retargeting efforts, doing yet another target can expose flaws in the retargeting technology, e.g., in lcc's code-generation interface. More interestingly, retargeting to a VM can expose design flaws in the VM itself.

Targeting lcc to MSIL amounts to writing a new back end for lcc that emits MSIL, and using the .NET tools to generate the appropriate .NET executable file. Writing the back end requires understanding both MSIL and the lcc code-generation interface.

The short summary is that lcc's MSIL back end—dubbed lcc.NET—supports all of Standard C except `setjmp` and `longjmp` and some uses of pointers to functions without prototypes. A new MSIL "linker" was required to cope with initialization and separate compilation, but this tool also helps detect programming errors. Lcc.NET also exposed some weak points in the lcc code-generation interface.

The Common Intermediate Language

MSIL is a stack-based virtual machine. While it is possible to interpret MSIL programs, there is no supported MSIL interpreter. Programs are translated into machine code by a just-in-time (JIT) code generator that is built into the .NET runtime system. MSIL programs are verified to be type-safe before execution, unless security settings are set otherwise.

MSIL supports the data types summarized in Table 1. The second column shows the corresponding C type. Managed pointers point to garbage-collectible objects and arrays and to their fields and elements. Transient pointers point to, for example, local variables and exist only on the evaluation stack. C pointers are, of course, represented by unmanaged pointers. The I, U, and R types denote the native integral and floating-point types. While lcc doesn't use values of these types explicitly, such values do appear on the evaluation stack as intermediate results, which impacts the code generator, as detailed below.

There are 255 instructions divided roughly into two groups: basic instructions, which are abstractions of the instructions found on most modern processors, and object-oriented instructions, which perform target-independent

Table 1. MSIL types.

Type Code	*C Types*	*Details*
I, U		signed/unsigned native integer of unspecified size
I1, I2, I4, I8	`signed char`, `short`, `int`, `long`,	signed/unsigned integers of specific sizes
U1, U2, U4, U8	`unsigned char`, `short`, `int`, `long`	
R4, R8, R8	`float`, `double`, `long double`	floating-point values of specific sizes
F		native floating-point with precision ≥ R8
U	*T* *	unmanaged pointers
&		managed pointers
O		opaque object references

object manipulation. There are also instructions for object creation and initialization and for exception handling. Table 3 in the Appendix lists all of the MSIL instructions.

Unlike the Java VM, MSIL does not have type-specific instructions for most of its operations, e.g., `add`, because the JIT code generator can determine the type from the contents of the evaluation stack.

Predictably, few of the object-oriented instructions are needed for implementing C. Indeed, MSIL's support for unmanaged pointers, arithmetic types of specified sizes, and indirect addressing makes it possible to implement unsafe languages like C.

THE LCC CODE-GENERATION INTERFACE

Lcc is distributed with back ends for the SPARC, MIPS, X86, and ALPHA for several platforms, and others have written back ends for additional platforms. Lcc is, by design, a monolithic compiler: Its back ends are loaded with the front end to form a single executable. A small code-generation interface defines the interaction between lcc's target-independent front end and its target-dependent back ends [4,5]. This interface consists of a few shared data structures, a 33-operator tree intermediate representation, or IR, that represents executable code, and 18 functions that manipulate and modify the shared data structures.

The shared data structures include tree nodes (`Node`), symbol-table entries (`Symbol`), and types (`Type`). The 33 tree IR operators are listed in Figure 1, and are explained in detail in Reference 5. Each generic operator can be specialized by appending an operand type suffix and a size in bytes. The 6 type suffixes are:

`F`	float	`P`	pointer
`I`	integer	`V`	void
`U`	unsigned	`B`	'block' (aggregate)

There can be up to 9 sizes. For example, `ADDF4` denotes a 4-byte floating addition, and `CVII2` denotes a conversion from an integer to a 2-byte integer. Not all of the 33 × 6 × 9 = 1782 operator combinations are meaningful, and the number of sizes on most targets is limited. On 32-bit targets, there are 130 type- and size-specific operators. Conversions on 32-bit targets, for instance, convert only between 4 and 4- or 8-byte floats, or widen or narrow between 3 sizes of integers. Some operators have only one or a few valid suffixes; for instance, the address operators (`ADDRL`, `ADDRF`, `ADDRG`) can have only the `P` type suffix and whatever size is the size of a pointer on the target. Back ends need accommodate only those type- and size-specific operators that are meaningful on their target.

```
CNST   ARG    ASGN   INDIR  CVF    CVI    CVP    CVU
NEG    CALL   RET    ADDRG  ADDRF  ADDRL  ADD    SUB
LSH    MOD    RSH    BAND   BCOM   BOR    BXOR   DIV
MUL    EQ     GE     GT     LE     LT     NE     JUMP   LABEL
```

Figure 1. Lcc tree IR generic operators.

Figure 2 summarizes the 18 code-generation functions. On most targets, many of these routines are very short, perhaps only a few calls to `printf`, because they simply emit assembly language. Most of the work is done in `gen`, `emit`, and `function`, which collaborate to generate and emit code for a function.

`void progbeg(int, char *[])`	initialize the back end
`void progend(void)`	finalize the back end
`void defsymbol(Symbol)`	initialize a symbol-table entry
`void export(Symbol)`	export a symbol
`void import(Symbol)`	import a symbol
`void global(Symbol)`	define a global
`void local(Symbol)`	define a local
`void address(Symbol, Symbol, long)`	define an address relative to a symbol
`void blockbeg(Env *)`	open a block-level scope
`void blockend(Env *)`	close a block-level scope
`void function(Symbol, Symbol [], Symbol [], int)`	define a function body
`void gen(Node)`	generate code
`void emit(Node)`	emit code
`void defconst(int, int, Value)`	initialize an arithmetic constant
`void defaddress(Symbol)`	initialize an address constant
`void defstring(int, char *)`	initialize a string constant
`void space(int)`	define an uninitialized block
`void segment(int)`	switch logical segments

Figure 2. Lcc code-generation functions.

For a given target, type sizes and pointers to the interface functions are packaged in an "interface record", which is selected at runtime by a command-line option. Lcc is thus a cross compiler, since it can emit code for any of the resident back ends. To lcc, MSIL is just another back end. Reference 8 gives more details on lcc's packaging, and Reference 5 is the complete specification of the code-generation interface.

Compilation Pipeline

In traditional environments, the lcc driver compiles a C program by preprocessing the source, compiling the preprocessed source into assembly language, and calling the assembler to generate object code. As shown in Figure 3, a linker combines object files and optional libraries into an executable file (Figure 3 omits the preprocessor). For example, on Microsoft Windows, the command

```
lcc -o hello.exe main.c hello.c
```

runs the equivalent of the following commands.

```
cpp ... main.c main.i
rcc -target=x86/win32 main.i main.asm
ml ... -Fomain.obj main.asm
cpp ... hello.c hello.i
rcc -target=x86/win32 hello.i hello.asm
ml ... -Fohello.obj hello.asm
link ... main.obj hello.obj -OUT:hello.exe liblcc.lib libc.lib
```

Cpp is the preprocessor, rcc is the compiler proper, ml is the assembler, and link is the linker. Also, readable names are used for the intermediate files above; in practice, the lcc driver generates temporary files.

There are no object files in .NET. A .NET executable program is packaged as an *assembly* that holds the MSIL code and metadata, which contains version, type, dependency, locale, and cryptographic information. Assemblies often exist as single executable files or a dynamically linked library, but multi-file assemblies are possible. Lcc.NET generates only single-file assemblies. Figure 4 shows the lcc.NET compilation pipeline. The lcc command shown above runs the following commands.

```
cpp ... main.c main.i                     preprocess main.c
rcc ... -target=msil/win32 main.i main.il compile main.i
cpp ... hello.c hello.i                   preprocess hello.c
rcc ... -target=msil/win32 hello.i hello.il
                                          compile hello.i
```

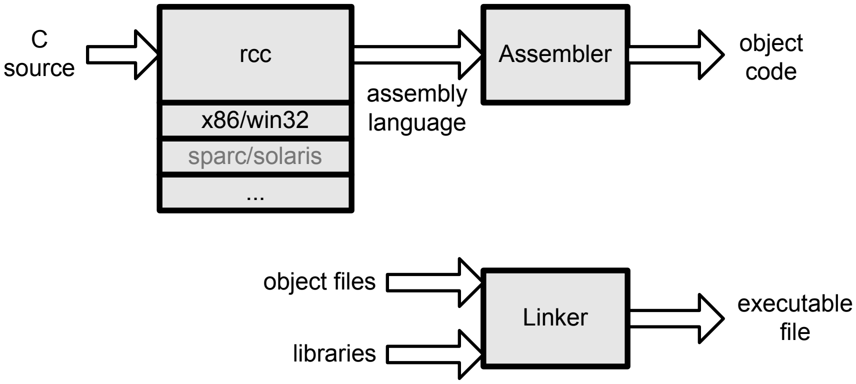

Figure 3. Lcc compilation pipeline in traditional environments.

```
illink ... main.il hello.il -l lcclib.dll -l mscvrt.dll
                                    link, search libraries lcclib.dll
                                    and msvcrt.dll, generate _assembly.il
ilasm ... -out:hello.exe main.il hello.il _assembly.il
                                    assemble and emit executable
```

Here, compilation ends with MSIL assembly language files, but these are unsuitable for execution because, for example, they can contain unresolved references. The lcc.NET linker, illink, reads these files, resolves references and performs some additional code-generation tasks, as detailed below, and writes an "entry point" file, named `_assembly.il` above, which contains assembly information and the main program. The .NET assembler, ilasm [10], functions both as an assembler and as a traditional linker. It reads the *same* MSIL files as illink along with the generated entry point file and emits an assembly as a .NET executable program. Lcc.NET relies on a small library of lcc-specific functions (`lcclib.dll`) and on the Standard C library that comes with Visual C .NET (`msvcrt.dll`).

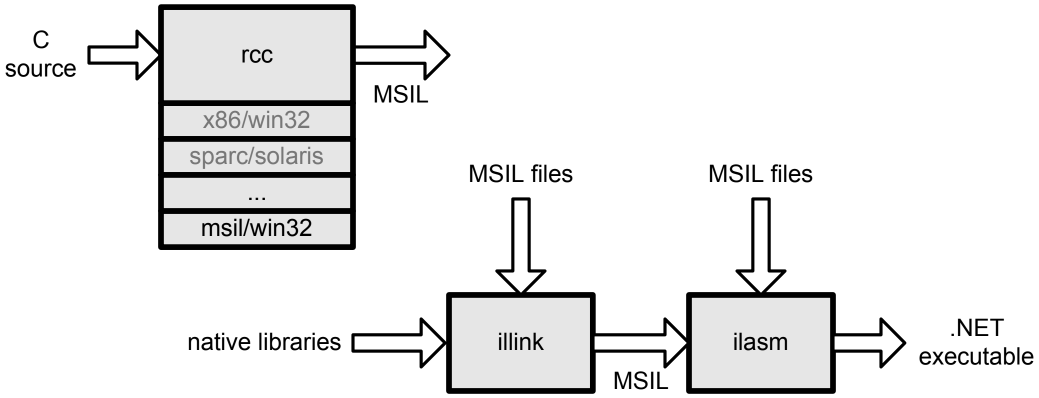

Figure 4. Lcc.NET compilation pipeline.

Mapping C onto MSIL

MSIL supports only classes, and classes contain static and instance members, which include fields and methods. For example, the following complete MSIL program prints "Hello world".

```
.assembly hello {}
.class public Hello {
.method static public void Main() cil managed {
        .entrypoint
        .maxstack 1
        ldstr "Hello world!"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
}
```

The `.assembly` declaration declares the name of this assembly. The `.class` declaration defines the class `Hello` and opens a scope in which to define its members. The `.method` declaration defines the static method `Main`, and `.entrypoint` stipulates that this method is the entry point. Methods must specify, using `.maxstack`, the maximum number of evaluation stack slots they use. The next three MSIL instructions push the Unicode string `Hello world!` on the stack, calls the `WriteLine` method in the .NET Frameworks library, and returns. Dimmed keywords are shown only for completeness and may otherwise be ignored.

Ilasm supports a single, unnamed global class whose members are those fields and methods defined outside of any class scope. Only static members are permitted. Lcc maps C globals onto static global fields and C functions onto static global methods. C parameters and locals are mapped to method parameters and locals. Most of C's data types map directly onto the corresponding MSIL data types as shown in Table 1, with two exceptions. The built-in MSIL array type cannot be used for C-style arrays because MSIL arrays are allocated in a managed heap. MSIL supports only Unicode strings that are also heap allocated and are not terminated by null characters. Lcc uses "value classes" of the appropriate number of bytes for arrays and for strings of 8-bit characters. A value class is a class whose instances appear only on the evaluation stack or as the value of fields in objects allocated dynamically.

Although ilasm reads multiple MSIL files, it treats them logically as a single input that defines classes and their members. Consequently, there is a single name space for top-level names. The names of C globals can be used as-is, but statics, including compiler-generated constants, must have globally unique names. For each separately compiled input, lcc generates a unique prefix from the current time and process number and inserts this prefix at the beginning of those names. Another alternative would be to use the nested class facilty to encapsulate the names from each compilation unit, but this approach would still require a unique name for the outer class and would require that class name in references to globals from other compilation units, which complicates linking because the linker would have to rewrite its input MSIL files.

For the prototypical C program

```
#include <stdio.h>
void main(void) { printf("Hello world!\n"); }
```

lcc emits

```
.class private value explicit ansi sealed 'int8[]' { .pack 1 .size 14 }
.method public hidebysig static void 'main'() cil managed
.maxstack 2
ldsflda valuetype 'int8[]' $3dc1b116_d70__2
call vararg int32 'printf'(void*)
pop
$L1:
ret
}
.field public static valuetype 'int8[]' $3dc1b116_d70__2 at $3dc1b116_d70_4
.data $3dc1b116_d70_4 = {
bytearray ( 48 65 6c 6c 6f 20 77 6f 72 6c 64 21 a 0 )
}
```

The `.class` declaration defines the type of the constant `"Hello world!\n"`, which is just a 14-byte sequence. The type name—`int8[]`—could be anything, but lcc uses C-like declarations for type names to make the generated code easier to decipher for MSIL programmers. Also, the size of the value type, 14 in the example above, is not part of the type. So, the class `int8[]` can be used as the type for all character arrays, including those allocated in the heap at runtime; similar comments apply to other array types. The `.field` and `.data` declarations collaborate to define a static field and initialize it to `"Hello world!\n"`. The unique prefix is `3dc1b116_d70_`, which appears in the field name and in the `.data` label. The method for `main` loads the address of the constant (`ldsflda`), calls `printf` (`call`), discards `printf`'s return value (`pop`), and returns (`ret`).

Note that the MSIL `call` instruction requires a full type signature—argument types and return type—for the callee. For functions with prototypes, lcc emits the appropriate signature; for those without prototypes, lcc fabricates a signature from the promoted types of the actual arguments; e.g., if the actual argument is a char, the promoted type is int, etc. A similar procedure is used when compiling function definitions into method declarations. Signatures are also required in the definitions of fields and locals and in all instructions for which the operand type cannot be determined from context, such as loads and stores; the signature of the `ldsflda` instruction above is an example. Perhaps a quarter of the code in the MSIL back end is devoted to forming and emitting type signatures.

Most of the interface functions and the instruction selection code for the MSIL back end are straightforward. Many of the lcc IR operators map directly onto corresponding MSIL instructions. Static initialization, function pointers, address arithmetic, and floating-point operators, and variable length argument lists proved to be the trouble spots.

Static Initialization

As the example above shows, MSIL supports static initialization of scalars and of sequences of scalars; it's also possible to initialize pointers, e.g., for the input

```
int x, *p = &x;
```

lcc emits

```
.field public static void* 'p' at $p
.data $p = {
&($x)
}
.field public static int32 'x' at $x
.data $x = { … }
```

The unique prefix and the type declarations are omitted in this and in subsequent displays to make them more readable. Address arithmetic can appear in constant pointer initializations. For instance,

```
int x[10], *p[2] = { 0, &x[5] };
```

initializes `p[1]` to the address of the 6th element of `x`. For constant address arithmetic of the form *x*+*n*, where *x* is a location and *n* is a byte offset, lcc generates a symbol to represent the location. It calls the `address` code-generation function (see Figure 2) with the new, partially initialized symbol, *x*, and *n*, and the back end completes the target-dependent initialization of the new symbol. For `x[5]`, the Intel x86 back end just sets the symbol name to "`x+20`", which the assembler accepts and computes the address intended.

MSIL and ilasm do not have similar facility. The only way to initialize such pointers is at execution time, so the code-generation function `defaddress` in the MSIL back end generates an initialization method that contains the appropriate address computations and assignments. For the initialization of `p[1]` above, lcc emits the following data and initialization method. The line numbers are for explanatory purposes and are not part of the emitted code.

```
    .field public static valuetype 'void*[]' 'p' at $p
    .data $p = {
    int32 (0),
    int32 (0)
    }
    .field public static valuetype 'int32[]' 'x' at $x
    .data $x = { … }
    …
1   .method public hidebysig static void $$_init() cil managed {
2   .maxstack 3
3   ldsflda valuetype 'void*[]' 'p'
4   ldc.i4 4
5   add
6   ldsflda valuetype 'int32[]' 'x'
7   ldc.i4 20
8   add
9   stind.i4
10  ret
    }
11  //$$INIT call void $$_init()
```

The initialization method appears in lines 1–10. Lines 3–5 compute the address of `p[1]`, lines 6–8 compute `&x[5]`, and line 9 uses indirection to store that value into `p[1]`. Line 11 directs illink to arrange for the initialization method to be called at program startup as detailed below.

This initialization problem forced the only change in the lcc code-generation interface: The `address` code generation function is optional. If it's omitted, lcc's front end generates code for address computations. This change is upward compatible with existing lcc back ends, so only the front end was modified.

Function Pointers

Initializations for function pointers cause simpler problems—emitting explicit assignments is the only way to initialize them. For example, given the code

```
static int say(const char *msg) { printf(msg); }
int (*hello)(const char *) = say;
```

lcc emits the assignment into the initialization method:

```
    .method public hidebysig static int32 $1_say(void*) cil managed {
    ...
    }
    .field public static method int32 *(void*) 'hello' at $hello
    .data $hello = {
    int32 (0)
    }
    .method public hidebysig static void $$_init() cil managed {
    .maxstack 3
1   ldsflda method int32 *(void*) 'hello'
2   ldftn int32 $1_say(void*)
3   stind.i4
    ret
    }
    //$$INIT call void $$_init()
```

Lines 1–3 stores the address of the `say` method indirectly into the `hello` field.

One of .NET's strong suits is inter-operability. It has built-in support for calls from managed code written in MSIL to unmanaged code in C or other languages that compile to native code and vice versa. The runtime system handles argument marshalling and calling sequence conversion on-the-fly when necessary. This facility makes it possible for lcc.NET to use the Microsoft Standard C library. Function pointers, however, are not supported fully, so the MSIL back end must do extra work.

In the general case, lcc must know if a function pointer is the address of a managed method or an unmanaged function. To simplify the implementation, lcc assumes that all function pointers are to managed code and converts those that might point to unmanaged code to managed equivalents at runtime. For example, if the code above is changed to

```
int (*hello)(const char *) = puts;
```

the body of the initialization method becomes

```
1    ldsflda method int32 *(void*) 'hello'
2    ldftn int32 'puts'(void*)
3    ldsfld int8 __is_unmanaged_puts
4    brfalse $L1
5    call void* __getMUThunk(void*)
6    $L1:
7    stind.i4
```

Lines 3-6 determine if puts is in unmanaged code during execution. The address of puts passed to __getMUThunk if __is_unmanaged_puts is true. __getMUThunk generates a “transition thunk” on-the-fly at runtime to handle calls to puts from managed code and returns a pointer to this thunk. Function pointers returned by unmanaged functions are also filtered through __getMUThunk.

There is also a __getUMThunk, which generates transition thunks at runtime for indirect calls from unmanaged code back to managed code. Calls to the Standard C library function qsort exemplify this variant of the problem. Consider

```
int compare(const void *x, const void *y) {
    return *(int*)x - *(int*)y;
}
    ...
    qsort(array, elemsize, nelems, compare);
    ...
```

When the call to qsort is compiled, lcc cannot determine if qsort is managed or unmanaged code, but it does know that compare is managed code. Lcc generates the following code for the call to qsort.

```
1    ldsflda valuetype 'int32[]' 'array'
     ...
4    ldftn int32 'compare'(void*,void*)
5    ldsfld int8 __is_unmanaged_qsort
6    brfalse $L16
7    call void* __getUMThunk(void*)
8    $L16:
9    call void 'qsort'(void*,int32,int32,method int32 *(void*,void*))
```

Lines 5–8 cause the pointer to compare to be passed to __getUMThunk if __is_unmanaged_qsort is true. __getUMThunk generates and returns the address of a thunk that makes it possible to call the managed compare from unmanaged code. Using these functions and flags, lcc insures that only unmanaged function pointers are passed to unmanaged code, and likewise for managed code.

__getMUThunk and __getUMThunk cache thunks so each thunk is generated only once. These short functions (80 lines) are part of lcclib.dll, the small lcc-specific library, which is written in C.

Function pointer support in lcc.NET has two limitations. C permits pointers to functions without prototypes to be assigned to pointers with prototypes, e.g.,

```
extern int puts();
int (*hello)(const char *) = puts;
```

is valid. In .NET, however, method “names” include signatures, so the name of puts is actually 'puts'(), which will not match 'puts'(void*) and thus causes an undefined reference diagnostic during linking. The second, more serious, limitation is that the transition thunks cannot handle pointers to unmanaged functions with a variable number of arguments, because there is no way to determine the number of actual arguments reliably. Thus printf cannot be used in place of puts in the examples above, for instance. Forthunately, these limitations rarely occur in practice with Standard C code.

Linking

Lcc emits an MSIL assembly language file for each input source file. It cannot, however, determine if external functions refer to functions defined in other MSIL modules or in unmanaged native-code libraries. As suggested in

Figure 4, the linker illink consumes MSIL files and names of libraries, determines which externals refer to unmanaged library functions, and writes an “entry-point” MSIL file, named `_assembly.il` in the examples above, that contains the appropriate declarations. In addition, illink emits into this file the assembly packaging required in the .NET environment, which includes the program entry point. The input MSIL files and the generated entry-point file are then passed to ilasm, which writes the final executable. Given the two files,

```
main.c:
    void main(void) {
        extern int (*hello)(char *);
    (*hello)("Hello world!\n");
    }
hello.c:
    static int say(const char *msg) { printf(msg); }
    int (*hello)(const char *) = say;
```

the command shown above,

```
lcc -o hello.exe main.c hello.c
```

generates the entry-point file shown in Figure 5.

```
 1   .assembly 'a.exe' {}
 2   .method public hidebysig static void $$INIT() cil managed {
 3   call void $$_init()
 4   ret
     }
 5   .method public hidebysig static void $Main(string[] argv) cil managed
{
 6   .entrypoint
 7   .maxstack 3
 8   call void $$INIT()
 9   call void 'main'()
10   ldc.i4 0
11   call void exit(int32)
12   ret
     }
13   .method public hidebysig static pinvokeimpl("msvcrt.dll" ansi cdecl)
         void 'exit'(int32) native unmanaged preservesig {}
14   .method public hidebysig static pinvokeimpl("msvcrt.dll" ansi cdecl)
         int32 'printf'(void*) native unmanaged preservesig {}
```

Figure 5. Entry-point file generated by illink.

Illink is a simple variant of the machine-independent linker described in Reference 3. It builds and manipulates three sets of symbols. Direct method calls and `ldftn` instructions contribute symbols to *R*, the set of functions referenced, and method definitions contribute symbols to *D*, the set of functions defined. Libraries are searched for symbols in $R - D$; those that are found are added to *D* and to *E*, the set of external functions defined in unmanaged code. Thus, placement of libraries in the illink command line is important, because searching a library can change both *R* and *D*.

At the end of linking, illink emits a method declaration for each symbol in *E* as exemplified by lines 13 and 14 in Figure 5. Line 14, for example, specifies that `printf` is an external static function defined in `msvcrt.dll`, the C runtime library. The `pinvokeimpl` attribute specifies that `printf` must be invoked using a platform-specific calling sequence, specifically, the C calling sequence (`cdecl`).

Illink also collaborates with lcc to generate initialization functions; it collects comment lines of the form

`//$$`*name instruction*

and emits *instruction* into the definition for the method `$$`*name* and arranges to call `$$`*name* before `main`. As explained above, lcc emits per-file initialization functions and lines like

```
//$$INIT call void $$__init()
```

which causes the initialization functions to be called at program start-up when `$$INIT` is called. In Figure 5, lines 2-4 define `$$INIT`, and it's called in line 8.

Finally, illink defines and initializes the `__is_unmanaged_`*name* flags emitted by lcc. It collects the flags as it reads the MSIL inputs and emits initialized field definitions for them into the entry-point file. For example, when `qsort` comes from the C library, illink emits

```
.field public static int8 __is_unmanaged_qsort at $_qsort
.data $_qsort = int8 (1)
```

Of course, illink could rewrite the MSIL code to avoid the runtime tests and to move the initialization code into the entry file, but in the current design, illink only reads the MSIL files generated by lcc; it does not write them.

Floating Point

Storage locations that hold floating-point values are of fixed size, either 4 or 8 bytes (see Table 1). Floating-point arguments, locals, return values, and values on the evaluation stack are represented using the internal F type. The precision of this "native" floating-point type is unspecified, but it must be at least that of R8. The F type allows a .NET implementation to use the most natural and efficient representation for floating-point numbers, e.g., the 80-bit extended precision on the Intel x86.

This design avoids some of the criticisms leveled at Java's floating-point semantics [9], which does not give the programmer full control of floating-point operations as specified by the IEEE 754 Standard. But the .NET design also complicates the back end. Lcc maps C arguments and locals onto MSIL arguments and locals. The JIT code generator may assign registers to some of these variables, which forces the back end to emit explicit conversions to insure that the precision of the values matches those of the types of the variables. The following C program illustrates the potential pitfall.

```
void main(void) {
    float temp = 0.0F, one = 1.0F, delta = 1.0F;
    while(temp != one) {
        temp = one + delta;
        printf(".");
        delta = delta/2.0F;
    }
    printf("%e is the least number that can be added to 1.0F\n", delta*4.0F);
}
```

This program computes and prints the smallest floating-point number that can be added to 1.0F, which is $1.192093*10^{-7}$. For the assignment to `temp`, lcc might emit

```
.locals ([0] float32 'delta')
.locals ([1] float32 'temp')
.locals ([2] float32 'one')
...
ldloc 2          temp = one + delta
ldloc 0
add
stloc 1
```

This program works—most of the time. If, however, the emboldened call to `printf` is removed, the program prints $2.220446*10^{-16}$. The Intel x86 JIT code generator assigns `temp` to a floating-point register, which holds an 80-bit value. The call to `printf` causes this register to be spilled to memory, and this spill includes a narrowing from 80 to 32 bits, changing the precision of the result. Without the call, `temp` is never narrowed to 32 bits, and the loop doesn't terminate at the proper point. Lcc must inject explicit narrowing operations to insure that variables have their declared precision; for the example above, lcc must emit

```
ldloc 2          temp = one + delta
ldloc 0
add
conv.r4
stloc 1
```

The `conv.r4` instruction narrows the sum to 32 bits before it's stored in `temp`, and the program works with or without the call to `printf`. Lcc injects these kinds of conversions when passing arguments, for assignments to locals and formals, and for return values.

Variable Length Argument Lists

MSIL has instructions designed specifically for dealing with variable length argument lists. While these instructions complicate the back end a bit, they also provide an important diagnostic benefit. For example,

```
#include <stdio.h>
#include <stdarg.h>
void print(char *fmt, ...) {
    va_list ap;                     __va_list ap
    va_start(ap, fmt);              __va_start(&ap)
    { int x = va_arg(ap, int);      *(int*)__va_arg(&ap,__typecode(int))
    double y = va_arg(ap, double);  ...similar translation...
    printf(fmt, x, y, va_arg(ap, char *)); }...similar translation...
    va_end(ap);                     ((void)0)
}

void main(void) {
    print("%d %e%s", 1, 2.3, "\n");
}
```

prints `1 2.300000e+000`. If, say, `23` is passed in place of the emboldened `2.3`, a runtime errors occurs, because the .NET runtime detects that the actual argument, `23`, is not a double. Conventional C environments do not detect these kinds of common errors and usually print gibberish, because they simply interpret the incoming bit pattern as a double. Invalid pointer arguments in variable length argument lists usually cause unannounced crashes. The .NET runtime also provides stack traces for all errors, so even dereferencing a null pointer is announced with a trace.

The MSIL back end collaborates with the macros `va_list`, `va_start`, `va_arg`, and `va_end` defined in `stdarg.h` to emit the appropriate MSIL instructions. The italicized fragments on the right above show the result of the non-trivial expansions; the expansions for the other two uses of `va_arg` are similar to the one shown. The MSIL back end recognizes the `__va` names as special built-in functions and emits inline code for them. The compile-time built-in `__typecode` returns the internal lcc code for its argument type. Standard C reserves the underscore prefix for exactly these kinds of uses. For the three lines annotated above, lcc emits

```
1   .locals ([0] valuetype [mscorlib]System.ArgIterator 'ap')__va_list ap
    .locals ([1] int32 'x')
    .locals ([3] void* '1')
    ...
2   ldloca 0                  __va_start(&ap)
3   arglist
4   call instance void [mscorlib]System.ArgIterator::
        .ctor(valuetype [mscorlib]System.RuntimeArgumentHandle)
5   ldloca 0                  int x = *(int*)__va_arg(&ap,__typecode(int))
6   call instance typedref [mscorlib]System.ArgIterator::GetNextArg()
7   refanyval int32
8   stloc 3
```

```
 9  ldloc 3
10  ldind.i4
11  stloc 1
```

Line 1 declares ap as an instance of the runtime system's argument-list type, ArgIterator, and lines 2–4 initialize ap to print's argument list. The arglist instruction is one of the special MSIL instructions for variable length argument lists; it retrieves a "handle" to the argument list of the method in which it appears. Lines 5–6 fetch the address of the next argument as a "typed reference", which is essentially a typed pointer, and the refanyval instruction in line 7 retrieves the actual address of the argument and verifies that the referent is of the indicated type. The refanyval instruction is where type errors like the one mentioned above are detected. Lines 8–11 store the actual address in a temporary, fetch it again, load the integer at that address, and store it in x.

The MSIL inter-operability facilities supports passing argument list handles between managed and unmanaged code, so the three statements surrounded by braces in print above can be replaced by

```
vprintf(fmt, ap);
```

and an appropriate unmanaged va_list value is passed to vprintf in the C library.

Measurements

While performance was a secondary goal in lcc.NET, preliminary measurements do give an estimate of the execution costs of using MSIL. Lcc compiles itself, and it emits similar code on all its targets, except for minor target-specific details in the back ends. It thus provides an apples-to-apples comparison: lcc compiled by lcc to a native executable compared to lcc compiled by lcc to MSIL in a .NET executable. Table 2 shows the execution speeds for these two instances of lcc compiling combine.c, a 7277-line module from GCC, the GNU C compiler. Numerous other inputs of varying sizes gave similar ratios of the execution times. The second column is the time for the native lcc and the third column is the time for lcc.NET given as JIT code generation time plus execution time. The JIT times were obtained from a sampling profiler. These times are for running only the compiler proper, and were taken on a 933MHz Intel Pentium III with 512MB of memory running Windows XP Professional Service Pack 1 and the retail version (build 3705) of the Microsoft .NET Framework.

Table 2. Sample execution times in seconds for lcc and lcc.NET on a 7277-line input.

Target	*lcc*	*lcc.NET*
null	0.22s	0.43+0.31=0.74s
x86/win32	0.44s	0.69+0.77=1.46s
msil/win32	1.07s	0.42+1.32=1.74s

The first column is the target option: The null target emits no output, and the x86/win32 and msil/win32 targets emit Intel x86 assembly language and MSIL. The null target executes only the code in front end and does no output and so gives a lower bound on lcc's execution speed. Once the MSIL is turned into x86 code by the JIT code generator, lcc.NET is just under two times slower than the native version, and this ratio remains essentially the same when emitting x86 assembly language. The MSIL back end must buffer the output of data initializations, which it does using lists of formatted strings. It thus uses more memory than the other back ends, which perhaps explains the smaller performance difference between the two compilers when emitting MSIL.

Including the JIT time, which cannot be avoided, makes lcc.NET about three times slower than the native version. The x86/win32 JIT time is much longer than the msil/win32 JIT time because the x86/win32 back end is much bigger—over 5600 lines compared to 842 lines. Most of the x86/win32 back end is generated automatically [4] and includes over 1400 lines of static initializations and one 3300-line function.

Typical back ends for lcc are small, at least when compared to other compilers. The MSIL back end itself is small, but it's only part of the solution. The number of non-blank, non-comment lines in lcc.NET components are as follows.

842	MSIL back end (C)
461	illink (C#)
125	Source for `lcclib.dll` (C)
46	MSIL-specific part of the lcc driver
118	MSIL-specific library header files

Discussion

Most virtual machines are designed to support only one language or perhaps a family of languages. The Java VM is perhaps the most recent design best known, but using virtual machines in programming language implementations has a long history [12]. MSIL is designed to support a wide range of languages, but it's clearly aimed more directly at modern, type-safe, object-oriented languages. Lcc.NET does demonstrate success at one end of that range, but it also exposes some MSIL weaknesses. Other languages with static initializations, address arithmetic, and separate compilation are likely to have porting problems similar to lcc, because these areas caused the most problems.

There are other areas that the lcc port does not stress but that may cause problems for some languages, particularly functional languages. For example, there is no facility in MSIL for direct manipulation of stack frames or return addresses, which complicate porting languages with coroutines, continuations, or language-level threads, for example. This omission is also why lcc.NET cannot support `setjmp` and `longjmp`. MSIL does support tail calls, which helps languages like Scheme that require them. While MSIL takes multi-language support further than other virtual machines, it is not a complete solution to the language half of the UNCOL problem.

Lcc's code-generation interface can claim some of the blame for the problems porting lcc to MSIL. This interface—now over a decade old—is spartan by design, and it was designed to use the capabilities of typical assemblers. Wherever possible, the lcc front end does extra work if the result simplified the code-generation interface. Likewise, there are few options in the interface; choice was traded for simplicity at every turn. Switch statements are an example. Lcc compiles a switch statement into a binary search of dense branch tables [4]. It emits code for the binary search and static tables of labels for the branch tables. MSIL has no facility for branch tables, so lcc.NET cannot emit them. Luckily, lcc can emit degenerate branch tables, which amounts to a binary search. MSIL does have a switch instruction, but lcc can't use it, because there's no code-generation interface function for switch statements. A more flexible, but complicated, interface design might include an optional switch statement interface function and use the current approach when it was omitted.

Static initializations are another example. The C Standard does not require static initialization; it requires only that initialization occur before execution begins. The lcc front end could generate code to perform the initializations much in the same way as the MSIL back does, but in a target-independent fashion. The code-generation interface could include an option to select this approach, for example. Doing so would simplify the MSIL port and ports to other targets with similar restrictions or limited assemblers.

Ports to "exotic" targets are valuable, in part, because they often expose errors in well established interfaces, and lcc is no exception; for example, modifying lcc to emit ASDL files exposed symbol-management errors [8]. Lcc can generate target-independent code to detect null pointer dereferences and to accumulate execution counts for performance monitoring. Selecting any of these options caused the front end to call code-generation functions in the wrong order; specifically, it called `defsymbol` before `progbeg`, which must be called first to initialize the back end (see Figure 2). Lcc's production back ends have only trivial `progend`s, which were unaffected by this error, but the MSIL back end creates some tables that are queried by its other functions, including `defsymbol`.

Acknowledgements

Thanks to Chris Fraser, Ronald Laeremans, Erik Meijer, Jim Miller, and Todd Proebsting for their comments, suggestions, and assistance with the finer points of the .NET environment. Serge Lidin responded with amazing speed to my problems with ilasm.

Lcc.NET is available at http://www.research.microsoft.com/downloads/; lcc 4.2 is available at http://www.cs.princeton.edu/software/lcc/.

References

1. ECMA International, 'Common Language Infrastructure (CLI)', Standard ECMA-335, Geneva, Dec. 2001. http://www.ecma.ch/ecma1/STAND/ecma-335.htm.

2. J. R. Ellis and D. L. Detlefs, ‘Safe, Efficient Garbage Collection for C++’, Technical Report CSL-93-4, Xerox PARC , Palo Alto, CA, June 1993. ftp://parcftp.xerox.com/pub/ellis/gc/gc.ps.

3. C. W. Fraser and D. R. Hanson, ‘A Machine-Independent Linker’, *Software—Practice & Experience*, **12** (4), 351–366 (1982).

4. C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, Menlo Park, CA, 1995.

5. C. W. Fraser and D. R. Hanson, ‘The lcc 4.x Code-Generation Interface’, *Technical Report MSR-TR-2001-64*, Microsoft Research, Redmond, WA, July 2001. http://www.research.microsoft.com/~drh/pubs/interface4.pdf.

6. J. Gough, *Compiling for the .NET Common Language Runtime (CLR)*, Prentice-Hall PTR, Upper Saddle River, NJ, 2002.

7. E. Gunnerson, *A Programmer's Introduction to C#*, second edition, Apress, Berkeley, CA, 2001.

8. D. R. Hanson, ‘Early Experience with ASDL in lcc’, *Software—Practice & Experience*, **29** (5), 417–435 (1999).

9. W. Kahan and J. D. Darcy, ‘How Java’s Floating-Point Hurts Everyone Everywhere’, A*CM 1998 Workshop on Java for High–Performance Network Computing*, invited talk, Mar. 1998. http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.

10. S. Lidin, *Inside Microsoft .NET IL Assembler*, Microsoft Press, Redmond, WA. 2002.

11. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, second edition, Addison-Wesley, Reading, MA, 1990.

12. M. C. Newey, P. C. Poole and W. M. Waite, ‘Abstract Machine Modelling to Produce Portable Software’, *Software—Practice & Experience*, **2** (2), 107–136 (1972).

13. D. S. Platt, *Introducing Microsoft .NET*, second edition, Microsoft Press, Redmond, WA. 2002.

14. T. B. Steel, Jr., ‘A First Version of UNCOL’, *Proceedings Western Joint Computer Conference*, May 1961, pp. 371–378.

APPENDIX: MSIL INSTRUCTIONS

Table 3 lists all of the MSIL instructions. The second column gives the stack transition for each instruction; the top of the stack is to the right of each list.

Table 3. MSIL instructions.

Instruction	*Stack Transition*	*Operation*
`add`	…, *value1, value2* ⇒ …, *value1* + *value2*	addition
`add.ovf`		checked addition
`add.ovr.un`		checked unsigned addition
`and`	…, *value1, value2* ⇒ …, *value1* & *value2*	bitwise AND
`arglist`	…, ⇒ …, *handle*	get argument list handle
`beq` *target*	…, *value1, value2* ⇒ …	branch if equal
`bge` *target*		branch if greater than or equal
`bge.un` *target*		branch if greater than or equal unsigned
`bgt` *target*		branch if greater than
`bgt.un` *target*		branch if greater than unsigned
`ble` *target*		branch if less than or equal
`ble.un` *target*		branch if less than or equal unsigned
`blt` *target*		branch if less than
`blt.un` *target*		branch if less than unsigned
`bne.un` *target*		branch if not equal unsigned
`box` *type*	…, *address* ⇒ …, *object*	box value type
`br` *target*	… ⇒ …	unconditional branch
`break`	… ⇒ …	breakpoint
`brfalse` *target*	…, *value* ⇒ …	branch if zero
`brtrue` *target*		branch if nonzero
`call` *method*	$\ldots, arg_1 \ldots arg_n \Rightarrow \ldots, value$	call
`calli` *signature*	$\ldots, arg_1 \ldots arg_n, address \Rightarrow \ldots, value$	call indirect
`callvirt` *method*	$\ldots, object, arg_1 \ldots arg_n \Rightarrow \ldots, value$	call virtual method
`castclass` *type*	…, *object* ⇒ …, *object*	cast object
`ceq`	…, *value1, value2* ⇒ …, 0 or 1	compare equal
`cgt`		compare greater than
`cgt.un`		compare greater than unsigned
`clt`		compare less than
`clt.un`		compare less than unsigned
`ckfinite`	…, *value* ⇒ …, *value*	check if finite
`conv.i`	…, *value* ⇒ …, *result*	convert to I, push I
`conv.i1`		convert to I1, push I4
`conv.i2`		convert to I2, push I4
`conv.i4`		convert to I4, push I4
`conv.i8`		convert to I8, push I4
`conv.r.un`		convert unsigned to F, push F
`conv.r4`		convert to R4, push F
`conv.r8`		convert to R8, push F
`conv.u`		convert to U, push U
`conv.u1`		convert to U1, push U4
`conv.u2`		convert to U2, push U4
`conv.u4`		convert to U4, push U4
`conv.u8`		convert to U8, push U8

Table 3. MSIL instructions.

Instruction	*Stack Transition*	*Operation*

`conv.ovf.i`
`conv.ovf.i1`
`conv.ovf.i2`
`conv.ovf.i4`
`conv.ovf.i8`
`conv.ovf.u`
`conv.ovf.u1`
`conv.ovf.u2`
`conv.ovf.u4`
`conv.ovf.u8`
	…, *value* ⇒ …, *result*	checked conversions (see above)

`conv.ovf.i.un`
`conv.ovf.i1.un`
`conv.ovf.i2.un`
`conv.ovf.i4.un`
`conv.ovf.i8.un`
`conv.ovf.u.un`
`conv.ovf.u1.un`
`conv.ovf.u2.un`
`conv.ovf.u4.un`
`conv.ovf.u8.un`
	…, *value* ⇒ …, *result*	checked conversion from unsigned (see above)

`cpblk`	…, *dstaddress, srcaddress, size* ⇒ …	copy memory
`cpobj` *type*	…, *dstaddress, srcaddress* ⇒ …	copy value type
`div` `div.un`	…, *value1, value2* ⇒ …, *value1 / value2*	division unsigned division
`dup`	…, *value* ⇒ …, *value, value*	duplicate
`endfilter`	…, *value* ⇒ …	end of exception handling filter
`endfinally`	… ⇒ …	end of finally clause
`initblk`	…, *address, value, size* ⇒ …	initialize memory
`initobj` *type*	…, *address* ⇒ …	initialize a value type
`isinst` *type*	…, *object* ⇒ …, *objec*t or null	type test and cast
`jmp` *method*	… ⇒ …	jump to method
`ldarg` *n*	… ⇒ …, *value*	push argument
`ldarga` *n*	… ⇒ …, *address*	push argument address
`ldc.i4` *constant* `ldc.i8` *constant* `ldc.r4` *constant* `ldc.r8` *constant*	… ⇒ …, *value*	push constant

`ldelem.i`
`ldelem.i1`
`ldelem.i2`
`ldelem.i4`
`ldelem.i8`
`ldelem.r4`
`ldelem.r8`
`ldelem.ref`
`ldelem.u1`
`ldelem.u2`
	…, *object, index* ⇒ …, *value*	push array element

`ldelema` *type*	…, *object, index* ⇒ …, *address*	push array element address
`ldfld` *field*	…, *object* ⇒ …, *value*	push instance ffield
`ldflda` *field*	…, *object* ⇒ …, *address*	push field address
`ldftn` *method*	… ⇒ …, *address*	push method address

Table 3. MSIL instructions.

Instruction	*Stack Transition*	*Operation*

`ldind.i`
`ldind.i1`
`ldind.i2`
`ldind.i4`
`ldind.i8`
`ldind.r4`
`ldind.r8`
`ldind.ref`
`ldind.u1`
`ldind.u2`
`ldind.u4`
	…, *address* ⇒ …, *value*	push indirect

`ldlen`	…, *address* ⇒ …, *value*	push array length
`ldloc` *n*	… ⇒ …, *value*	push local variable
`ldloca` *n*	… ⇒ …, *address*	push local variable address
`ldnull`	… ⇒ …, *null*	push null
`ldobj` *type*	…, *address* ⇒ …, *value*	push value type
`ldsfld` *field*	… ⇒ …, *value*	push static field
`ldsflda` *field*	… ⇒ …, *address*	push static field address
`ldstr` *string*	… ⇒ …, *object*	push literal string
`ldtoken` *token*	… ⇒ …, *handler*	push metadata handler
`ldvirtftn` *method*	… ⇒ …, *address*	push virtual method address
`leave` *target*	… ⇒ …	branch out of protected region
`localloc`	…, *size* ⇒ …, *address*	allocate local storage
`mkrefany` *type*	…, *address* ⇒ …, *typed reference*	push typed reference

`mul`
`mul.ovf`
`mul.ovf.un`
	…, *value1, value2* ⇒ …, *value1* * *value2*	multiplication
		checked multiplication
		checked unsigned multiplication

`neg`	…, *value* ⇒ …, *–value*	negation
`newarr` *type*	…, *size* ⇒ …, *object*	create array
`newobj` *method*	…, arg_1 … arg_n ⇒ …, *object*	create object and call constructor
`nop`	… ⇒ …	no operation
`not`	…, *value* ⇒ …, ¬*value*	bitwise complement
`or`	…, *value1, value2* ⇒ …, *value1* | *value2*	bitwise inclusive OR
`pop`	…, *value* ⇒ …	discard value
`refanytype` `refanyval` *type*	…, *typed reference* ⇒ …, *type* …, *typed reference* ⇒ …, *address*	retrieve type token from typed reference retrieve address from typed reference
`rem` `rem.un`	…, *value1, value2* ⇒ …, *value1* % *value2*	remainder unsigned remainder
`ret`	… ⇒ … …, *value* (callee) ⇒ …, *value* (caller)	return
`rethrow`	… ⇒ …	rethrow exception
`shl`	…, *value1, value2* ⇒ …, *value1* << *value2*	left shift
`shr` `shr.un`	…, *value1, value2* ⇒ …, *value1* >> *value2*	arithmetic right shift logical right shift
`sizeof` *type*	…, ⇒ …, *size*	push size of value type
`starg` *n*	…, *value* ⇒ …	store to argument

Table 3. MSIL instructions.

Instruction	*Stack Transition*	*Operation*

`stelem.i`
`stelem.i1`
`stelem.i2`
`stelem.i4`
`stelem.i8`
`stelem.r4`
`stelem.ref`
`stelem.r8`
	…, *object*, *index*, *value* ⇒ …	store to array element

`stfld` *field*	…, *object*, *value* ⇒ …	store to instance field

`stind.i`
`stind.i1`
`stind.i2`
`stind.i4`
`stind.i8`
`stind.r4`
`stind.r8`
`stind.ref`
	…, *address*, *value* ⇒ …	store indirect

`stloc` *n*	…, *value* ⇒ …	store to local variable
`stobj` *type*	…, *address*, *value* ⇒ …	store value type
`stsfld` *field*	…, *value* ⇒ …	store to static field

`sub`
`sub.ovf`
`sub.ovf.un`
	…, *value1, value2* ⇒ …, *value1* – *value2*	subtraction
		checked subtraction
		checked unsigned subtraction

`switch` *n*, off_1, …, off_n	…, *value* ⇒ …	computed branch
`tail.`	… ⇒ …	next call is a tail call
`throw`	…, *object* ⇒ …	throw exception
`unaligned.`	…, *address* ⇒ …, *address*	next instruction uses unaligned address
`unbox` *type*	…, *object* ⇒ …, *address*	unbox value type
`volatile.`	…, *address* ⇒ …, *address*	next instruction uses volatile address
`xor`	…, *value1, value2* ⇒ …, *value1* ^ *value2*	bitwise exclusive OR