

Intermediate Language

**Tomáš Matoušek
&
Ladislav Prošek**

tmd.havit.cz/teaching/csharp.htm

IL Assembler

- language for writing metadata and IL code
- full power of CLR
 - other languages are less powerful
- structurally a simple language
 - program structure corresponds to metadata logical structure
 - assembly level, global code level, class level, method level
 - metadata knowledge necessary
 - many keywords
 - many details
- metadata declarations
 - more or less correspond to metadata tables
 - a little bit of abstraction

Naming Conventions

- simple names
 - identifier or single quoted literal (e.g. 'My Precious')
 - reserved identifiers
 - keywords, .ctor, .cctor
- composite names
 - [resolution scope]Namespace.Type/NestedType::Member
 - resolution scope
 - AssemblyName
 - .module ModuleName

Visibility

- type visibility
 - private (internal in C#)
 - public (public in C#)

- member visibility

IL

public

private

family

assembly

famorassem

famandassem

privatescope

C#

public

private

protected

internal

internal protected

N/A

N/A

- nested type visibility
 - nested private, nested family, nested assembly, ...

Types

- **primitives**
 - void, bool, char, object, string
 - [unsigned] int{8 | 16 | 32 | 64}
 - native [unsigned] int
 - float{32 | 64}
- **pointers, vectors, arrays**
 - unmanaged pointer: type*
 - managed pointer: type&
 - vector: type[]
 - arrays: type[,], type[2...5], type[0..., 4...]
- **native types**
 - interoperation with unmanaged code
 - describes how managed types are marshaled to native ones
 - lpstr (pointer to zero terminated ANSI string), lpwstr, bstr, ...
 - iunknown, idispatch
 - ...

CLR VM Instructions

- operating on evaluation stack
- stack comprises of typed slots
 - can contain managed references, structures, pointers (&, *)
- instructions not typed
 - the stack is
 - e.g. single instruction for addition
- bytecode
 - opcode is 1B <opcode> or 2B wide <0xfe><opcode>
 - an instruction can have a parameter (token or integer)
- long vs. short forms of instruction having integer parameter
 - long: no suffix, parameter is 4B wide
 - short: .s suffix, parameter is 1B wide

Instruction Set

- flow control
- data loading, storing, and addressing
- operations on vector
- arithmetic operations
- operations on classes and structures
- stack operations
- memory block operations
- ...

Flow Control Instructions

- intra-procedure branching
 - unconditional
 - br
 - conditional
 - brfalse, brtrue
 - comparative
 - beq, bne, bge, bgt, ble, blt
 - switch(N, label[0], ..., label[N-1])
- inter-procedure branching
 - ret
 - jmp
 - abandon the current method and jump to another one
 - call, callvirt, calli
 - a tail call with suffix tail.
- exception handling
 - throw, rethrow, leave, endfilter, endfinally

Loading and Storing

- constant loading
 - load a constant on the top of the stack (TOS)
 - ldc.i4, ldc.i4.{m1, 0, ..., 8}
 - ldnull
- string loading
 - ldstr
- indirect loading
 - loads a value stored on the address
 - ldind.{i1, i2, i4, i8, u1, u2, u4, i, r4, r8, ref}
- indirect storing
 - stores a value on the address
 - stind.{i1, i2, i4, i8, u1, u2, u4, i, r4, r8, ref}

Fields, Locals, Arguments

- locals
 - load: ldloc, ldloc.{0, 1, 2, 3}
 - store: stloc
 - address: ldloca
- arguments
 - load: ldarg, ldarg.{0, 1, 2, 3, 4}
 - store: starg
 - address: ldarga
- fields (instance, static)
 - load: ldfld, ldsfld
 - store: stfld, stsfld
 - address: ldflda, ldsflda

Vectors

- construction: newarr
- length: lrlen
- elements
 - load: ldelem.{i1, i2, i4, i8, u1, u2, u4, i, r4, r8, ref}
 - store: stelem.{i1, i2, i4, i8, r4, r8, i, ref}
 - address: ldelema

Classes and Structures

- classes
 - newobj ... creates a new instance and calls a specified constructor
 - castclass ... changes a type of reference on the TOS
 - isinst ... checks whether the TOS is of the specified type
- structures
 - initobj ... initializes a structure with zeroes or null references
 - sizeof ... loads a size of a structure
 - box ... converts a structure to an object
 - unbox ... converts an object (a boxed structure) to the structure

 - ldobj ... loads a structure on the specified address on the TOS
 - stobj ... stores a structure on the TOS to the specified address
 - cpobj ... copies a structure from source address to dest. address

Arithmetic Operations

- add, sub, mul, div, rem, neg
 - also with prefix .ovf
 - applicable only on integers
 - overflow causes an exception (recall: checked operations)
- shift (shl, shr)
- bitwise (and, or, xor, not)
- conversions (conv.{i1, i2, i4, i8, u1, u2, u4, i, u, r4, r8})
 - narrowing, widening
 - integer conversions also with prefix .ovf
- condition checks (ceq, cgt, clt, ckfinite)

Miscellaneous

- `nop` ... no operation
- `dup` ... duplicates the top of the stack
- `pop` ... removes the top of the stack
- `break` ... debugging breakpoint

- `ldftn` ... loads an address of a non-virtual function entry point
- `ldvirtftn` ... ditto for virtual methods

- `cpblk` ... copies a block of memory (`memcpy`)
- `initblk` ... fills a block of memory with a value (`memset`)
- `localloc` ... allocates a block of memory on a stack

- `ldtoken` ... loads a run-time representation of a token
- `mkrefany`, `refanytype`, `refanyval` ... discussed later

Fields Layout

- defines how fields are laid out
- important for interoperability with unmanaged code
- layouts
 - auto
 - CLR reorders fields to make GC faster
 - managed references precede other fields
 - sequential
 - fields are laid out as they were declared
 - explicit
 - user specifies offsets
 - fields can overlap (useful for defining unions)
 - managed references cannot be overlapped

```
[StructLayout(LayoutKind.Explicit)]  
public struct MyUnion  
{  
    [FieldOffset(0)] uint number;  
    [FieldOffset(0)] ushort lo;  
    [FieldOffset(2)] ushort hi;  
}
```

```
.class public sealed explicit value MyUnion  
{  
    .field[0] unsigned int32 number  
    .field[0] unsigned int16 lo  
    .field[2] unsigned int16 hi  
}
```

VarArgs Methods

- how to pass variable number of arguments
 - creating an array of objects
 - implicit array creation if params keyword used in C#
 - vararg method modifier

- declaration

```
.method public vararg string Format(string format)
{ /* body */ }
```

- invocation

- the sentinel precedes actual argument types in call signature

```
ldstr "(%d;%d) "
ldc.i4.1
ldc.i4.2
call vararg string Format(string, ..., int32, int32)
```

System.TypedReference

- a structure containing
 - a pointer
 - a runtime type handle
- embeds the type information along with the pointer
- can only be the type of a local variable or an argument

- instructions manipulating typed reference
 - mkrefany <type token>
 - converts a pointer of a specified type to a typed reference
 - refanytype
 - extract a type token from a typed reference
 - refanyval <type token>
 - extracts an address from a typed reference
 - the value has to be of a specified type

Iterating Arguments

- structure `System.ArgIterator`
 - allows iterating through arguments on the stack
 - implemented in EE
 - methods
 - `GetNextArg ...` returns typed reference to the next argument
 - `GetRemainingCount ...` returns remaining argument count
- IL instruction `arglist` returns structure `System.RuntimeArgumentHandle`
- `System.ArgIterator` constructor takes that handle

VarArgs via Undocumented C#

C#	IL
TypedReference <code>__makeref</code> (storage)	<code>ldloca.s <storage></code> <code>mkrefany <type token of storage></code>
T <code>__refvalue</code> (TypedReference, T)	<code><load arg. of type TypedReference></code> <code>refanyval <type token T></code>
Type <code>__reftype</code> (TypedReference)	<code><load arg. of type TypedReference></code> <code>refanytype</code> <code>call class Type</code> <code>Type::GetTypeFromHandle(RuntimeTypeHandle)</code>
<pre>public void M(int a, __arglist) { ArgIterator args; args = new ArgIterator(__arglist); }</pre>	<pre>.method public vararg void M(int32 a) { .locals valuetype ArgIterator args ldloca args arglist call instance void ArgIterator::.ctor(RuntimeArgumentHandle) }</pre>
<code>M(1, __arglist(1, "x"));</code>	<code>call instance void M(int32, ..., int32, string);</code>

P/Invoke and IJW

- P/Invoke

- `.method pinvokeimpl(<mapping>) { }`
- `<mapping> ::= <module> [as <name>] [flags]`
- e.g.

```
.method static pinvokeimpl("kernel32.dll" as "Beep")  
    bool MyBeep(int freq, int duration) { }
```

- IJW (It Just Works)

- methods containing native code
- cannot be handled by ILASM nor ILDASM
- no round-tripping

```
.method public static void* malloc(unsigned int32) native unmanaged  
{ /* embedded native code */ }
```

Java Virtual Machine

- types
 - primitive
 - *byte (b), short (s), int (i), long (l), char (c), float (f), double (d), boolean*
 - reference (*a*)
 - *class type, interface type, array (vector or jagged array)*
 - missing
 - pointers, structures, delegates, events, properties, multidim. arrays
- instructions
 - typed (e.g. *{i, l, f, d}add*)
 - almost a subset of CLR VM
- evaluation stack (*operand stack*)
 - 32-bit slots
 - longs and doubles stored in two slots
- local variables
 - stored in array of 32-bit wide items (like evaluation stack)

JVM Instruction Set

- stack: pop, pop2, dup, dup2, **swap**
- constants: {T}ipush, {T}const, ldc
- locals: {T}load, {T}store

- instantiation: new, newarray, anewarray, multianewarray
- classes: instanceof, checkcast
- arrays: {T}aload, {T}astore, arraylength
- fields: getfield, putfield, getstatic, putstatic

- arithmetic: **{T}inc**, {T}add, {T}sub, {T}mul, {T}div, {T}rem, {T}neg
- shifts: {T}shl, {T}shr, {T}ushr
- bitwise: {T}and, {T}or, {T}xor
- comparisons: {T}cmp, {T}cmpl, {T}cmpg
- conversions: {i, l, f}2{l, f, d}
- conditional branching: if{eq, lt, le, ne, gt, ge, null, nonnull}, {T}cmp{op}
- branching: goto, ret, {T}return
- switching: tableswitch, lookupswitch
- calls: invokevirtual, invokeinterface, invokespecial, invokestatic
- exceptions: athrow, jsr
- monitors: **monitorenter**, **monitorexit**

CLR vs. JVM

- JVM has not
 - multi-language support
 - structures, unions
 - pointers (neither managed nor unmanaged) \Rightarrow no ref/out parameters
 - method pointers \Rightarrow no delegates, events
 - native code interoperability features
 - remoting support (transparent proxies, method interceptors)
 - multidimensional arrays
 - unchecked arithmetic operations
 - varargs
 - tail calls
 - declarative security
 - properties
 - support for generics (JVM 1.5 is not aware of generics)
- CLR has not
 - instructions feasible for interpretation

Bytecode Example: Object Construction

```
int i, j;  
new MyClass(i, j);
```

JVM:

```
new <id of MyClass>           // allocates uninitialized instance  
dup  
iload_1  
iload_2  
invokespecial <id of MyClass.<init>>
```

CLR:

```
ldloc.0  
ldloc.1  
newobj <token of instance void MyClass::.ctor(int32, int32)>
```

Bytecode Example: Loop

```
void spin() { for (int i = 0; i < 100; i++); }
```

JVM:

```
0:  iconst_0      // Push int constant
    istore_1     // Store into local variable 1 (i=0)
    goto 8      // First time through don't increment
5:  iinc 1 1     // Increment local variable 1 by 1 (i++)
8:  iload_1      // Push local variable 1 (i)
    bipush 100  // Push int constant
    icmplt 5    // Compare and loop if less than (i < 100)
e:  return      // Return void when done
```

CLR:

```
0:  ldc.i4.0
    stloc.0
    br.s 8
4:  ldloc.0
    ldc.i4.1
    add
    stloc.0
8:  ldloc.0
    ldc.i4.s 100
    blt.s 4
d:  ret
```

Tools

- ILASM
 - IL assembler
 - location: Windows\Microsoft.NET\Framework\{version}
- ILDASM
 - IL disassembler
 - location: ...\\Visual Studio .NET\\SDK\\{version}\\Bin
- PEVerify
 - IL verifier
 - location: ...\\Visual Studio .NET\\SDK\\{version}\\Bin
- Jbimp
 - converts Java bytecode to IL
 - location: ...\\Visual Studio .NET\\SDK\\{version}\\Bin
- Jad
 - Java bytecode disassembler (Pavel Kouznetsov)
 - <http://www.kpdus.com/jad.html>