

Intermediate Language Assembler

Tomáš Matoušek

tm.matfyz.cz/teaching/clr.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
 - simple names separated by dots
- full names
 - [resolution scope]Namespace.Type/NestedType::Member
 - resolution scope
 - AssemblyName
 - .module ModuleName
- name prefix – namespace (not a metadata item)

```
.namespace composite_name { declarations }
```

Manifest Declarations

- assembly definition

```
.assembly name
{
    .ver version
    .locale culture_name
    permissions
    custom attributes
}
```

- assembly reference

```
.assembly extern name [as alias]
{
    .ver version
    .locale culture_name
    .publickey = (bytes)
}
```

- module definition

```
.module name
    custom attributes
```

- module reference

```
.module extern name
```

- additional assembly file definition

```
.file [nometadata] name
```

- managed resources definition

```
.mresource [public|private] name
{
    [.assembly extern alias / .file name at offset]
}
```

Type Definition

```
.class flags simple_name[<generic_param_list>]
  [extends class_ref] [implements class_ref_list]
{
  [.param type generic_param_name custom attributes]*
  [.pack align]
  [.size size]
  custom attributes
  declarations
}
```

- **flags**
 - visibility, nesting
 - layout (auto, sequential, explicit)
 - type semantics (interface, abstract, sealed)
 - implementation (serializable, beforefieldinit)
 - pseudo-flags (value, enum)
- **notes**
 - *simple_name* for a generic type ends with ` {#type params}
 - type parameter list comprises of names and constrains
 - pack and size not valid for auto-layout
 - generic params referred to via `!{name}` or `!{ordinal}`

Fields and Data Items

- field definition

```
.field [[offset]] flags type simple_name [= default_value] [at data_label]  
custom attributes
```

- flags

- visibility
 - contract (static, initonly, literal, notserialized, specialname)
- notes
 - offset determines the field position within the explicitly-laid-out type
 - literal fields cannot be accessed directly, only via reflection (used in enums)
 - default value != initial value, default value can be accessed only via reflection

- static field mapping

- data label designates a named place in .sdata or .tls section the field is mapped to
- more fields can be mapped to a single memory address
- references and non-public fields disallowed to be mapped to
- data provided “as is”; no platform specific conversions (e.g. endianness)

- data definition

```
.data [tls] label = [ int32(value) | double(value) | bytearray(data) | ... ]
```

Fields Layout

- defines how fields are laid out
- important for interoperability with unmanaged code
- type layouts
 - auto
 - CLR reorders fields to make GC faster
 - managed references precede the 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  
}
```

Methods

- method definition

```
.method [flags] [conv] return_type name[<generic_param_list>] (arg_list) [impl_flags]
{
    custom attributes
    [.locals [init] (local_variables list)]*
    [.override method_ref]*
    [.entrypoint]
    [.param[index] [= default_value] custom_attributes]*
    IL code
    exception blocks
}
```

- flags
 - visibility
 - contract (static, final, virtual, newslot)
 - implementation (abstract, specialname, pinvokeimpl)
 - method implementation flags
 - cil, native, runtime, internalcall, synchronized, noinlining, ...
- call convention
 - instance, explicit
 - vararg, unmanaged {cdecl, stdcall, thiscall, fastcall}
- name
 - special reserved names: .ctor, .cctor
- generic params
 - referred to via !:{name} or !:{ordinal}

Method Groupings: Properties, Events

- property definition

```
.property type simple_name(param_type_list) [= default_value]
{
    [.set method_ref]           // void (param_type_list, type)
    [.get method_ref]          // type (param_type_list)
    [.other method_ref]*      // void ()
    custom attributes
}
```

- C#, VB.NET: `get_`, `set_` prefixes; `.other` semantics unused

- event definition

```
.event delegate_type simple_name
{
    .addon method_ref          // void (delegate_type)
    .removeon method_ref      // void (delegate_type)
    [.fire method_ref]        // void (delegate_type)
    [.other method_ref]*      // void ()
    custom attributes
}
```

- C#, VB.NET: `add_`, `remove_` prefixes; `.fire`, `.other` semantics unused

Custom Attributes

- adding custom attribute to a definition

```
.custom instance void class_ref::.ctor(arg_list) [= serialized_attribute_data]
```

- owner defined by attribute positioning
 - owner with scope (curly braces)
 - custom attributes defined within the scope

```
.assemblymscorlib  
{  
  .custom instance void System.CLSCompliantAttribute::.ctor(bool)  
    = (01 00 01 00 00)  
}
```

- without scope
 - immediately after

```
.field public static int32 MyThreadStaticField  
.custom instance void [mscorlib]System.ThreadStaticAttribute::.ctor()  
  = (01 00 00 00)
```

Visibility Modifiers

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

	<u>C#</u>
<u>IL</u>	
public	public
private	private
family	protected
assembly	internal
famorassem	internal protected
famandasse	N/A
privatescope	N/A
- **nested type visibility**
 - nested private, nested family, nested assembly, ...

VarArg Methods

- How to pass a variable number of arguments?
 - creating an array of objects
 - implicit array creation if params keyword used in C#
 - vararg method modifier
- declaration
- invocation
 - the “sentinel” (ellipsis) precedes actual argument types in the call

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

System.TypedReference

- a structure containing
 - a pointer
 - a 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

VarArgs via Undocumented C#

C#	IL
<pre>TypedReference __makeref (storage)</pre>	<pre>ldloca.s <storage> mkrefany <type token of storage></pre>
<pre>T __refvalue (TypedReference, T)</pre>	<pre><load arg. of type TypedReference> refanyval <type token T></pre>
<pre>Type __reftype (TypedReference)</pre>	<pre><load arg. of type TypedReference> refanytype call class Type Type::GetTypeFromHandle (RuntimeTypeHandle)</pre>
<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>
<pre>M (1, __arglist (1, "x"));</pre>	<pre>call instance void M (int32, ..., int32, string);</pre>

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

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 */ }
```

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 or 2B wide
 - an instruction can have parameter(s)
- 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
- vector operations
- arithmetic operations
- operations on classes and structures
- stack operations
- memory block operations
- ...

Flow Control Instructions

- intra-procedural branching
 - unconditional
 - br
 - conditional
 - brfalse, brtrue
 - comparative
 - beq, bne, bge, bgt, ble, blt
 - switch(N, label[0], ..., label[N-1])
- inter-procedural 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: `ldlen`
- 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: `ldlema`

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 ...` `TypedReference`