COMPILING FOR THE COMMON INTERMEDIATE LANGUAGE

The Common Intermediate Language

The Common Intermediate Language (CIL – also known as Microsoft Intermediate Language, or MSIL) is an intermediate representation developed by Microsoft to be the core language of the .NET framework. All the .NET languages such as C#, VB .NET of VC++ are compiled to CIL, which is the only language accepted by the Common Language Runtime (CLR) of .NET.[1]

CIL Instruction Set Architecture

The CIL is a stack-based architecture (having a bytecode format, like JAVA bytecode its instructions are also of varying lengths). As it was meant to support many features of advanced languages, it has many dedicated instructions, such as those needed to handle the creation and manipulation of objects and the invocation of object methods.

In this document we describe a subset of the language that corresponds to a basic stack-based architecture, but a useful subset that simple languages can be compiled to.

The columns in the next three sections show the assembly format, description and transition of the stack contents for the instructions in this subset. A typical stack transition is shown as: `..,val1,val2 -> ..,result`. This describes an instruction removing the top two values on the stack (val2 is the top value), and pushing the "result".

[1] For more detailed information, see: http://msdn.microsoft.com/net/ecma

CIL Instructions: Arithmetic Instructions

Assembly	*Description*	*Stack Transition*
`add`	adds two values on top of stack	`..,val1,val2 -> ..,result`
`and`	ands two values on top of stack	`..,val1,val2 -> ..,result`
`ceq`	compares top two stack values, pushes 1 if they are equal, 0 if else	`..,val1,val2 -> ..,result`
`cgt`	compares top two stack values, pushes 1 if val1 is greater than val2, 0 if else	`..,val1,val2 -> ..,result`
`clt`	compares top two stack values, pushes 1 if val1 is less than val2, 0 if else	`..,val1,val2 -> ..,result`
`div`	divides val1 by val2	`..,val1,val2 -> ..,result`
`dup`	duplicates top value on the stack	`..,val1 -> ..,val1,val1`
`mul`	multiplies two values on top of the stack	`..,val1,val2 -> ..,result`
`neg`	negates top value on the stack	`..,val1 -> ..,result`
`not`	bitwise complement of top value on stack	`..,val1 -> ..,result`
`or`	bitwise OR of two values on top of stack	`..,val1,val2 -> ..,result`
`rem`	remainder of val1 by val2 (val1 % val2)	`..,val1,val2 -> ..,result`
`shl`	Shift val1 left by val2 bits	`..,val1,val2 -> ..,result`
`shr`	shift val1 right by val2 bits	`..,val1,val2 -> ..,result`
`sub`	subtract val2 from val1	`..,val1,val2 -> ..,result`
`xor`	bitwise XOR of top two stack values	`..,val1,val2 -> ..,result`

CIL Instructions: Control Instructions

Assembly	*Description*	*Stack Transition*
`beq` *targ*	branch to targ if top stack values equal	`..,val1,val2 -> ..`
`bge` *targ*	branch to targ if val1 (2nd stack value) >= val2 (top stack value)	`..,val1,val2 -> ..`
`bgt` *targ*	branch to targ if val1 > val2	`..,val1,val2 -> ..`
`ble` *targ*	branch to targ if val1 <= val2	`..,val1,val2 -> ..`
`blt` *targ*	branch to targ if val1 < val2	`..,val1,val2 -> ..`
`br` *targ*	unconditional branch to target	`.. -> ..`
`brfalse` *targ*	branch to targ if val1 is false or null	`..,val1 -> ..`
`brtrue` *targ*	branch to targ if val1 is false or zero	`..,val1 -> ..`
`call` *method*	call a method	`..,arg1..argn -> ..,retval(if any)`
`nop`	no operation	`.. -> ..`
`ret`	return from method	`retval`[2] `-> ..,retval(if any)`

[2] The callee's stack should be empty except for the return value (if any), which is pushed to the top of the caller's stack.

CIL Instructions: Memory/Stack Instructions

Assembly	*Description*	*Stack Transition*
`ldarg` *num*	push numth argument onto top of the stack	.. -> ..,arg_{num}
`ldc.i4` *num*	push number num onto top of the stack	.. -> ..,num
`ldloc` *indx*	push indxth local variable onto top of the stack	.. -> ..,val1
`pop`	pop top value from the stack	..,val-> ..
stloc indx	pop top value from stack to indxth loc. variable	..,val1 -> ..

Compiling the ICS142 Language For CIL

Consider a simple while statement inside an if-statement, such as the one below. A CIL bytecode sequence that implements it (inefficiently, but correctly) is shown below it, along with the stack contents for each line in the next column.

```
{ int a = Read();
    if ((a > 0) && (a < 20)){
        while (a > 0){
           WriteHex(a);
           WriteLn();
           a = a - 1;
        }
    }
}
```

Bytecode assembly	*Stack contents after instruction*
{	$
call Test4::Read()	$a
stloc 1	$
ldloc 1	$a
ldc.i4 0	$a, 0
ble IL_B	$
ldloc 1	$a
ldc.i4 14	$a, 14
bgt IL_B	$
IL_A: ldloc 1	$a
ldc.i4 0	$a, 0
ble IL_B	$
ldloc 1	$a
call Wrapper::WriteHex(int32)	$
call Wrapper::WriteLn()	$
ldloc 1	$a
ldc.i4.1	$a, 1
sub	$(a-1)
stloc 1	$
br IL_A	$
IL_B: ret	$
}	

Important points:

Procedures accept arguments on the stack, with the first argument pushed first. On return all arguments will have been popped and replaced with a return value (if there is one).

Stack contents: At the return of a procedure there must be the same number of entries on the stack as in the entry, minus the number of arguments (if any), plus any return value (if any). This is another way of saying each procedure's stack has to be empty when returning from that procedure, with the possible presence of a return value on top of the stack.

Local Variables: CIL allows accessing up to 65535 local variables to push/pop from the stack. Instruction `stloc` *num* pops a value and loads it to local variable *num* (*num* = 0 .. 65534), and `ldloc` *num* pushes a value in local variable *num* to the top of the stack. While emitting code for procedures, you will have the opportunity to declare how many local variables each procedure will have.

Storing each assignment to a variable with the `stloc` instruction (and loading each usage of a variable in an expression with the `ldloc` instruction) should result code that automatically fulfills the stack contents constraint. While this is not very efficient, it saves the compiler from having to keeping track of stack state.

Using the Wrapper Class to Create Output Files

The output for this assignment will be in assembly file format. To produce output files, you will use the Wrapper class that will be made available as a DLL file through the TA's website. The Wrapper class makes the following public methods available:

```
public void    insertMain(string[] code);
public void    insertMain(string[] code,
                          int      localVarNum,
                          int      maxStackNum);
```

These methods insert a code sequence as the body of the Main() method. The first one assumes a maximum local variable number and stack size of 8, in the second one these are supplied explicitly. The CIL instructions are to be entered as an array of strings, preferably with one string per instruction (although strings of the form “ldc.i4 10\ndup\n” are also permissible).

```
public void    WriteFile(string fileName);
```

This method creates a CIL assembly file <fileName> with all the contents (a Main() and procedures) that have been entered so far.

```
public void    insertProcedure(string   procedureName,
                               string[] code,
                               bool     returnsVal,
                               int      argNum,
                               int      localVarNum,
                               int      maxStackNum);
```

This method is similar to insertMain(), except that it inserts a code sequence for a procedure <procedureName>, specifying the number of arguments (only of int32 type for out input language), whether a value is returned, and the maximum number of local variables and stack length. The instructions are entered as strings in an array.

```
public string  getProcedureCall(string procedureName,
                                 bool   returnsVal,
                                 int    argNum);
```

This method returns the string argument for a call instruction. It generates a valid argument recognizable by an assembler for a procedure with the given name, arguments and return type.

Predefined Methods

The following procedures are already built- in, you can insert calls to them from the body of your compiled CIL code with the getProcedureCall() method of the Wrapper class.

```
void  WriteLn();         // prints new line
void  Write(int32);      // prints an integer in decimal form
void  WriteHex(int32);   // prints an integer in hexadecimal form
int32 Read();            // reads an integer from the command line
```

An example of this is given here in this C# code segment.

```
string[] main = new string [4];
main[0] = "ldc.i4 10";
main[1] = getProcedureCall("WriteHex",false,1);
main[2] = getProcedureCall("WriteLn",false,0);
main[3] = "ret";
insertMain(main);
```

The output of this code is a CIL assembly segment for a Main() method:

```
.method static void Main() cil managed {
    .entrypoint
    .maxstack 8
    ldc.i4 10
    call void class Wrapper::WriteHex(int32)
    call void class Wrapper::WriteLn()
    ret
}
```