Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine

Lecture 25:

Interface With High-Level Language

Slide show prepared by Kip R. Irvine, Revision date: 07/07/2002

Modified on April 23.2005

by Dr. Nikolay Metodiev Sirakov

- Chapter corrections (Web)   Assembly language sources (Web)

Chapter Overview

- Why Link ASM and HLL Programs?
  - General and Calling Conventions
  - External Identifiers
- Inline Assembly Code
  - __asm Directive
  - File Encryption Example
- Linking to C++ Programs
  - Linking to Borland C++
  - ReadSector Example
- Special Section: Optimizing Your Code
  - Loop Optimization Example
  - FindArray Example
  - Creating the FindArray Project

Why Link ASM and HLL Programs?

- Use high-level language for overall project development
  - Relieves programmer from low-level details
- Use assembly language code
  - Speed up critical sections of code
  - Access nonstandard hardware devices
  - Write platform-specific code
  - Extend the HLL's capabilities

General Conventions

- Considerations when calling assembly language procedures from high-level languages:
  - Both must use the same naming convention (rules regarding the naming of variables and procedures)
  - Both must use the same memory model, with compatible segment names
  - Both must use the same calling convention

Calling Convention

- Identifies specific registers that must be preserved by procedures
- Determines how arguments are passed to procedures: in registers, on the stack, in shared memory, etc.
- Determines the order in which arguments are passed by calling programs to procedures
- Determines whether arguments are passed by value or by reference
- Determines how the stack pointer is restored after a procedure call
- Determines how functions return values

External Identifiers

- An external identifier is a name that has been placed in a module's object file in such a way that the linker can make the name available to other program modules.
- The linker resolves references to external identifiers, but can only do so if the same naming convention is used in all program modules.

Inline Assembly Code

- Assembly language source code that is inserted directly into a HLL program.
- Compilers such as Microsoft Visual C++ and Borland C++ have compiler-specific directives that identify inline ASM code.
- Efficient inline code executes quickly because CALL and RET instructions are not required.
- Simple to code because there are no external names, memory models, or naming conventions involved.

Disadvantage:

- Decidedly not portable because it is written for a single platform.

_asm Directive in Microsoft Visual C++

- Can be placed at the beginning of a single statement
- Or, It can mark the beginning of a block of assembly language statements
- Syntax:

```
__asm  statement

__asm {
   statement-1
   statement-2
   ...
   statement-n
}
```

Commenting Styles

All of the following comment styles are acceptable, but the latter two are preferred:

```
mov  esi,buf      ; initialize index register
mov  esi,buf     // initialize index register
mov  esi,buf     /* initialize index register */
```

You Can Do the Following . . .

- Use any instruction from the Intel instruction set
- Use register names as operands
- Reference function parameters by name
- Reference code labels and variables that were declared outside the asm block
- Use numeric literals that incorporate either assembler-style or C-style radix notation
- Use the PTR operator in statements such as

  inc BYTE PTR [esi]
- Use the EVEN and ALIGN directives
- Use LENGTH, TYPE, and SIZE directives

You Cannot Do the Following . . .

- Use data definition directives such as DB, DW, or BYTE
- Use assembler operators other than PTR
- Use STRUCT, RECORD, WIDTH, and MASK
- Use macro directives such as MACRO, REPT, IRC, IRP
- Reference segments by name.
  - (You can, however, use segment register names as operands.)

Register Usage

- In general, you can modify EAX, EBX, ECX, and EDX in your inline code because the compiler does not expect these values to be preserved between statements
- Conversely, always save and restore ESI, EDI, and EBP.

See the Inline Test demonstration program.

File Encryption Example

- Reads a file, encrypts it, and writes the output to another file.
- The TranslateBuffer function uses an __asm block to define statements that loop through a character array and XOR each character with a predefined value.

View the Encode2.cpp program listing

Linking Assembly Language to C++

- Basic Structure - Two Modules
  - The first module, written in assembly language, contains the external procedure
  - The second module contains the C/C++ code that starts and ends the program
- The C++ module adds the extern qualifier to the external assembly language function prototype.
- The "C" specifier must be included to prevent name decoration by the C++ compiler:

```
extern "C" functionName( parameterList );
```

Name Decoration

Also known as name mangling. HLL compilers do this to uniquely identify overloaded functions. A function such as:

```
int ArraySum( int * p, int count )
```

would be exported as a decorated name that encodes the return type, function name, and parameter types. For example:

```
int_ArraySum_pInt_int
```

C++ compilers vary in the way they decorate function names.

Special Section: Optimizing Your Code

- The 90/10 rule: 90% of a program's CPU time is spent executing 10% of the program's code
- We will concentrate on optimizing ASM code for speed of execution
- Loops are the most effective place to optimize code
- Two simple ways to optimize a loop:
  - Move invariant code out of the loop
  - Substitute registers for variables to reduce the number of memory accesses
  - Take advantage of high-level instructions such as XLAT, SCASB, and MOVSD.

Loop Optimization Example

- We will write a short program that calculates and displays the number of elapsed minutes, over a period of *n* days.
- The following variables are used:

```
.data
days DWORD ?
minutesInDay DWORD ?
totalMinutes DWORD ?
str1 BYTE "Daily total minutes: ",0
```

Sample Program Output

```
Daily total minutes: +1440
Daily total minutes: +2880
Daily total minutes: +4320
Daily total minutes: +5760
Daily total minutes: +7200
Daily total minutes: +8640
Daily total minutes: +10080
Daily total minutes: +11520
.
.
Daily total minutes: +67680
Daily total minutes: +69120
Daily total minutes: +70560
Daily total minutes: +72000
```

View the complete source code.

Version 1

No optimization.

```
    mov days,0
    mov totalMinutes,0

L1:                                 ; loop contains 15 instructions
    mov eax,24                      ; minutesInDay = 24 * 60
    mov ebx,60
    mul ebx
    mov minutesInDay,eax
    mov edx,totalMinutes            ; totalMinutes += minutesInDay
    add edx,minutesInDay
    mov totalMinutes,edx
    mov edx,OFFSET str1             ; "Daily total minutes: "
    call WriteString
    mov eax,totalMinutes            ; display totalMinutes
    call WriteInt
    call Crlf
    inc days                        ; days++
    cmp days,50                     ; if days < 50,
    jb  L1                          ; repeat the loop
```

Version 2

Move calculation of minutesInDay outside the loop, and assign EDX before the loop. The loop now contains 10 instructions.

```
     mov days,0
     mov totalMinutes,0
     mov eax,24                   ; minutesInDay = 24 * 60
     mov ebx,60
     mul ebx
     mov minutesInDay,eax
     mov edx,OFFSET str1          ; "Daily total minutes: "

L1:  mov edx,totalMinutes         ; totalMinutes += minutesInDay
     add edx,minutesInDay
     mov totalMinutes,edx
     call WriteString             ; display str1 (offset in EDX)
     mov eax,totalMinutes         ; display totalMinutes
     call WriteInt
     call Crlf
     inc days                     ; days++
     cmp days,50                  ; if days < 50,
     jb  L1                       ; repeat the loop
```

Version 3

Move totalMinutes to EAX, use EAX throughout loop. Use constant expresion for minutesInDay calculation. The loop now contains 7 instructions.

```
    C_minutesInDay = 24 * 60            ; constant expression
    mov days,0
    mov totalMinutes,0
    mov eax,totalMinutes
    mov edx,OFFSET str1 ; "Daily total minutes: "

L1: add eax,C_minutesInDay     ; totalMinutes += minutesInDay
    call WriteString           ; display str1 (offset in EDX)
    call WriteInt              ; display totalMinutes (EAX)
    call Crlf
    inc days                   ; days++
    cmp days,50                ; if days < 50,
    jb  L1                     ; repeat the loop

    mov totalMinutes,eax       ; update variable
```

Using Assembly Language to Optimize C++

- Find out how to make your C++ compiler produce an assembly language source listing
  - /FAs command-line option in Visual C++, for example
- Optimize loops for speed
- Use hardware-level I/O for optimum speed
- Use BIOS-level I/O for medium speed

FindArray Example

Let's write a C++ function that searches for the first matching integer in an array. The function returns true if the integer is found, and false if it is not:

```
#include "findarr.h"

bool FindArray( long searchVal, long array[],
                long count )
{
  for(int i = 0; i < count; i++)
    if( searchVal == array[i] )
      return true;
  return false;
}
```

Code Produced by C++ Compiler

optimization switch turned off (1 of 3)

```
_searchVal$ = 8
_array$ = 12
_count$ = 16
_i$ = -4

_FindArray PROC NEAR
; 29   : {
    push ebp
    mov  ebp, esp
    push ecx
; 30   :   for(int i = 0; i < count; i++)
    mov  DWORD PTR _i$[ebp], 0
    jmp  SHORT $L174
$L175:
    mov  eax, DWORD PTR _i$[ebp]
    add  eax, 1
    mov  DWORD PTR _i$[ebp], eax
```

Code Produced by C++ Compiler
(2 of 3)

```
$L174:
    mov  ecx, DWORD PTR _i$[ebp]
    cmp  ecx, DWORD PTR _count$[ebp]
    jge  SHORT $L176
; 31   : if( searchVal == array[i] )
    mov  edx, DWORD PTR _i$[ebp]
    mov  eax, DWORD PTR _array$[ebp]
    mov  ecx, DWORD PTR _searchVal$[ebp]
    cmp  ecx, DWORD PTR [eax+edx*4]
    jne  SHORT $L177
; 32   : return true;
    mov  al, 1
    jmp  SHORT $L172
$L177:
; 33   :
; 34   : return false;
    jmp  SHORT $L175
```

Code Produced by C++ Compiler
(3 of 3)

```
$L176:
    xor  al, al                ; AL = 0

$L172:
; 35   : }
    mov  esp, ebp              ; restore stack pointer
    pop  ebp
    ret  0
_FindArray ENDP
```

Hand-Coded Assembly Language (1 of 2)

```
true = 1
false = 0

; Stack parameters:
srchVal    equ   [ebp+08]
arrayPtr   equ   [ebp+12]
count      equ   [ebp+16]

.code
_FindArray PROC near
     push  ebp
     mov   ebp,esp
     push  edi

     mov   eax, srchVal            ; search value
     mov   ecx, count              ; number of items
     mov   edi, arrayPtr           ; pointer to array
```

Hand-Coded Assembly Language (2 of 2)

```
        repne scasd                  ; do the search
        jz    returnTrue             ; ZF = 1 if found

returnFalse:
        mov   al, false
        jmp   short exit

returnTrue:
        mov   al, true

exit:
        pop   edi
        pop   ebp
        ret
_FindArray ENDP
```

Creating the FindArray Project

(using Microsoft Visual Studio 6.0)

- Run Visual C++ and create a project named FindArray.
- Add a CPP source file to the project named main.cpp. This file should contain the C++ main() function that calls FindArray. View a sample.
- Add a new header file named FindArr.h to the project. This file contains the function prototype for FindArray. View a sample.
- Create a file named Scasd.asm and place it in the project directory. This file contains the source code for the FindArray procedure. View a sample.
- Use ML.EXE to assemble the Scasd.asm file, producing Scasd.obj. Do not try to link the program.
- Insert Scasd.obj into your C++ project. (Select *Add Files*... from the *Project* menu.) (this needs to be verified)
- Build and run the project.

Creating the FindArray Project
(using Microsoft Visual Studio)

- Run Visual C++.Net and create a new project named **FindArray**.
- Add a blank C++ source file to the project named main.cpp. Type the main() function that calls FindArray. View a sample.
- Add a new header file named FindArr.h to the project. This file contains the function prototype for FindArray. View a sample.
- Create a file named Scasd.asm and place it in the project directory. This file contains the source code for the FindArray procedure. View a sample.
- Use ML.EXE to assemble the Scasd.asm file, producing Scasd.obj. Do not try to link the program.
- Insert Scasd.obj into your C++ project.
- Build and run the project.