Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 13: High-Level Language Interface

Slide show prepared by the author

Revision date: 1/15/2014

Chapter Overview

- **Introduction**
- Inline Assembly Code
- Linking 32-Bit Assembly Language Code to C/C++

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.

What's Next

- Introduction
- **Inline Assembly Code**
- Linking 32-Bit Assembly Language Code to C/C++

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.
- 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 the OFFSET operator (but LEA is ok)
- 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

What's Next

- Introduction
- Inline Assembly Code
- **Linking 32-Bit Assembly Language Code to C/C++**

Linking Assembly Language to Visual 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

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
```

The problem with name decoration is that the C++

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

Summary

- Use assembly language top optimize sections of applications written in high-level languages
    - inline asm code
    - linked procedures
- Naming conventions, name decoration
- Calling convention determined by HLL program
- OK to call C functions from assembly language

The End