بسم الله الرّحمٰن الرّحيم

EE213 Computer organization and assembly language

Spring 2018

13

High-Level Language Interface

Outlines

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

introduction

•Where convenience and development time are more important than speed or code size High-level language is used

•Assembly language can be used for fine-tuning of such programs to speedup critical sections of code

- •Control high speed hardware
- •Memory-resident code
- •Access non-standard hardware devices
- •Write platform specific code
- •Extend the high-level language capabilities

•Interface or connection between high-level languages and assembly language

General Conventions

•There are a number of general considerations that must be addressed when calling assembly language procedures from high-level languages.

•**Naming Convention** refers to the rules or characteristics regarding the naming of variables and procedures

•**Memory model** used by a program determines the segment size and whether calls and references will be near or far

General Conventions

- **Calling Conventions**
- Refer to the low-level details about how procedures are called
- Which registers must be preserved by called procedures
- The method used to pass the arguments
- Arguments passing order
- Whether arguments are passed by value or by reference
- How the stack pointer is restored after a procedure call
- How functions return values to the calling program

General Conventions

- **External Identifiers:** Must have compatible naming conventions
  - A C-compiler adds an underscore to all names
  - Preserves the case

- **Segment Names:** When linking an assembly language procedure to a program written in a high-level language, segment names must be compatible.

- **Memory Models:** A calling program and the called procedure must both use the same memory model.

.model directive

- In 16-bit and 32-bit modes, MASM uses the **.MODEL** directive to determine several important characteristics of a program:
- its memory model type
- procedure naming scheme
- and parameter passing convention.

```
.MODEL memorymodel [, modeloptions]
```

Table 13-1 Memory Models.

Model	Description
Tiny	A single segment, containing both code and data. This model is used by programs having a .com extension in their filenames.
Small	One code segment and one data segment. All code and data are near, by default.
Medium	Multiple code segments and a single data segment.
Compact	One code segment and multiple data segments.
Large	Multiple code and data segments.
Huge	Same as the large model, except that individual data items may be larger than a single segment.
Flat	Protected mode. Uses 32-bit offsets for code and data. All data and code (including system resources) are in a single 32-bit segment.

- All of the models, with the exception of *flat*, are used when programming in 16-bit real-address mode

Model options

•The ModelOptions field in the .MODEL directive can contain both a language specifier and a stack distance.

```
.MODEL memorymodel [[, langtype]] [[,stackdistance]]
```

•**Stack Distance:**

•Specifying **NEARSTACK** groups the stack segment into a single physical segment along with data.

•The stack segment register (SS) is assumed to hold the same address as the data segment register (DS).

•**FARSTACK** does not group the stack; thus SS does not equal DS.

•Near subroutine calls and returns transfer control between procedures in the same code segment.

•Far subroutine calls and returns pass control between different segments.

Language specifiers

```
.model flat, C
.model flat, STDCALL
```

- **STDCALL** is the language specifier used when calling Windows system functions.
- It causes subroutine arguments to be pushed on the stack
- in reverse order.
- To remove the arguments from stack STDCALL requires a constant operand to be supplied in the RET instruction.
- The constant indicates the value added to ESP after the return address is popped from the stack by RET.

STDCALL

```
AddTwo PROC
       push ebp
       mov ebp,esp
       mov eax,[ebp + 12] ; second parameter
       add eax,[ebp + 8] ; first parameter
       pop ebp
       ret 8                ; clean up the stack
AddTwo ENDPP
```

•Finally, STDCALL modifies exported (public) procedure names by storing them in the following format:

```
       _name@nn
```

- •A leading underscore is added to the procedure name, and an integer follows the @ sign indicating the number of bytes used by the procedure parameters (rounded upward to a multiple of 4).

C specifer

•The C language specifier requires procedure arguments to be pushed on the stack from last to first, like STDCALL.
•Regarding the removal of arguments from the stack after a procedure call, the C language specifier places responsibility on the caller.
• In the calling program, a constant is added to ESP, resetting it to the value it had before the arguments were pushed:

```
push 6          ; second argument
push 5          ; first argument
call AddTwo
add esp,8       ; clean up the stack
```

•The C language specifier appends a leading underscore character to external procedure names. For example:

```
_AddTwo
```

Inline assembly code

- *Inline assembly code* is assembly language source code that is inserted directly into high-level language programs.
- **Advantage**: no *external linking issues*, *naming problems*, and parameter passing protocols to worry about.

- **Disadvantage**: lack of portability.
  - E.g. Inline assembly code that runs on an Intel Pentium processor will not run on a RISC processor.

The __asm Directive

•In Visual C++, the **__asm** directive can be placed at the beginning of a single statement, or it can mark the beginning of a block of assembly language statements:

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

Features

- Use most instructions from the x86 instruction set.
- Use register names as operands.
- Reference function parameters by name.
- Code labels and variables declared outside the __asm block are supported
- Numeric constants can be used that incorporate either assembler-style or C-style radix notation.
- For example, **0A26h** and **0xA26** are equivalent and can both be used
- Use the **PTR** operator can be used
- Use the EVEN and ALIGN directives

Limitations

- Cannot use data definition directives
- Cannot use assembler operators (other than `PTR`).
- Cannot reference segments by name.
- You cannot make any assumptions about **register values** at the beginning of an **asm** block.
  - The registers may have been modified by code that executed just before the asm block (e.g. **__fastcall**).
- Cannot use STRUCT, RECORD, WIDTH, and MASK.
- Cannot use macro directives, including MACRO, REPT, IRC, IRP, and ENDM, or macro operators (<>, !, &, %, and .TYPE).

Length, Type, and Size

With inline Assembly:

- The **LENGTH** operator returns the number of elements in an array.
- The **TYPE** operator returns one of the following:
  - The number of bytes used by a C or C++ type or scalar variable
  - The number of bytes used by a structure
  - For an array, the size of a single array element.
- The **SIZE** operator returns **LENGTH * TYPE.**

```
struct Package {
long originZip; // 4
long destinationZip; // 4
float shippingPrice; // 4
};
char myChar;
bool myBool;
short myShort;
int myInt;
long myLong;
float myFloat;
double myDouble;
Package myPackage;
long double myLongDouble;
long myLongArray[10];
```

```
__asm {
mov eax,myPackage.destinationZip;
mov eax,LENGTH myInt;     // 1
mov eax,LENGTH myLongArray; // 10
mov eax,TYPE myChar;              // 1
mov eax,TYPE myBool;              // 1
mov eax,TYPE myShort;    // 2
mov eax,TYPE myInt;               // 4
mov eax,TYPE myLong;     // 4
mov eax,TYPE myFloat;    // 4
mov eax,TYPE myDouble;   // 8
mov eax,TYPE myPackage;  // 12
mov eax,TYPE myLongDouble;        // 8
mov eax,TYPE myLongArray;         // 4
mov eax,SIZE myLong;     // 4
mov eax,SIZE myPackage;  // 12
mov eax,SIZE myLongArray;         // 40
}
```

Home Work

13.2.2 File Encryption Example