Chapter 12: High-Level Language Interface
Chapter Overview

- **Introduction**
- Inline Assembly Code
- C calls assembly procedures
- Assembly calls C procedures
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**
- C calls assembly procedures
- Assembly calls C procedures
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 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.)
char q[]={4, 5, 6};
void foo(void)
{
    char p[]={1, 2, 3};

    _asm{
        lea eax, p
        mov byte ptr [eax], 0

        mov ebx, offset q
        mov byte ptr [ebx], 0
    }
}

• Why can’t we use “mov eax, offset p”?
• Can we use “lea ebx, q”?
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.
Example:

```c
int *p, *q, count;

... 

// use assembly for fast memory copy
_asm{
    mov edi, p
    mov esi, q
    mov ecx, count
    cld 
    rep movsd
}
// the copy is done
```
What's Next

- Introduction
- Inline Assembly Code
- C calls assembly procedures
- Assembly calls C procedures
C calls Assembly Procedures

- Write performance-critical procedures as a standalone .asm file
- Can be called from C programs
- Assembly procedure must be declared before it can be called
C calls Assembly Procedures

• C calling convention
  – Arguments are pushed onto the stack in reverse order
  – Return value is stored in EAX
  – The caller is responsible to clean up the stack
    • i.e., to remove the arguments on the stack after a procedure call
C side

extern int ASM_add(int, int);

main()
{
    printf("%d", ASM_add(3, 5));
}
Assembly side

INCLUDE Irvine32.inc

.code

ASM_add PROC C,
    a:DWORD, b:DWORD

    mov eax,a
    add eax,b

    ret
ASM_add ENDP

END
What's Next

• Introduction
• Inline Assembly Code
• C calls assembly procedures
• Assembly calls C procedures
Assembly Calls C procedures

- Assembly is not a good choice to implement complicated control structures.
- Calls a C procedure to take over complex (but not performance-critical) tasks.
- Ex: calling a follow-up procedure from an interrupt service routine (ISR).
C side

int C_add(int x, int y)
{
    return x+y;
}

Assembly side

C_add PROTO C,
    x:DWORD, y:DWORD
.data
    a    DWORD ?
    b    DWORD ?
.code
MAIN PROC
    ...
    INVOKE C_add, a, b
    ...
MAIN ENDP
END
Fin