High-Level Language Interface

Computer Organization and Assembly Languages

Yung-Yu Chuang

2006/12/18

with slides by Kip Irvine

Why link ASM and HLL programs?

- Assembly is rarely used to develop the entire program.
- 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.
- 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 the **`LENGTH`**, **`SIZE`** and **`TYPE`** 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`**

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`**.
- You can't use **`OFFSET`**, but you can us **`LEA`** instruction to retrieve the offset of a variable.

```
        lea esi, buffer
```

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.

TranslateBuffer

```
void TranslateBuffer(char * buf,
                     unsigned count,
                     unsigned char eChar )
{
  __asm {
     mov esi,buf      ; set index register
     mov ecx,count    /* set loop counter */
     mov al,eChar
  L1:
     xor  [esi],al
     inc  esi
     Loop L1
 } // asm

}
```

File encryption

```
...
while (!infile.eof() )
{
  infile.read(buffer, BUFSIZE );
  count = infile.gcount();
  TranslateBuffer(buffer, count, encryptCode);
  outfile.write(buffer, count);
}
...
```

TranslateBuffer

```
   push ebp
   mov  ebp, esp
   sub  esp, 40h
   push ebx
   push esi
   push edi
   mov  esi,buf     ; set index register
   mov ecx,count   /* set loop counter */
   mov al,eChar
L1:
   xor  [esi],al
   inc  esi
   Loop L1
   pop  edi
   pop  esi
   pop  ebx
   mov  esp, ebp
   pop  ebp
```

File encryption

```
while (!infile.eof() )
 {
   infile.read(buffer, BUFSIZE );
   count = infile.gcount();
   __asm {             to avoid the calling overhead
      lea esi,buffer
      mov ecx,count
      mov al, encryptChar
   L1:
      xor [esi],al
      inc  esi
      Loop L1
  } // asm
   outfile.write(buffer, count);
 }
```