High-Level Language Interface

Computer Organization and Assembly Languages

Yung-Yu Chuang

2005/12/15

with slides by Kip Irvine

Overview

- Why Link ASM and HLL Programs?
- Inline Assembly Code
- Linking to C++ Programs
- Optimizing Your Code

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.
- 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

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.

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);
  }
```

File encryption

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

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
```

The problem with name decoration is that the C++ compiler assumes that your assembly language function's name is decorated. The C++ compiler tells the linker to look for a decorated name.

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
```

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
```

Version 4

Substitute ECX for the days variable. Remove initial assignments to days and totalMinutes.

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

L1:; loop contains 7 instructions
   add eax,C_minutesInDay ; totalMinutes+=minutesInDay
   call WriteString     ; display str1 (offset in EDX)
   call WriteInt        ; display totalMinutes (EAX)
   call Crlf
   inc ecx              ; days (ECX)++
   cmp ecx,50           ; if days < 50,
   jb  L1               ; repeat the loop
   mov totalMinutes,eax ; update variable
   mov days,ecx         ; update variable
```

Using assembly 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

```
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

```
     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
```