Lecture 18

- Assembly
- Linking and Loading
- Assembling Separate Files
- C++/Assembly Interface

Assembly Process (Review)

- Assembler translates symbolic assembly language program into numeric machine code.
  - translation is 1:1
  - everything in the source code must be converted into numeric code (labels, opcodes, register names, memory locations, immediates…)
- Ideally – want to scan a line, then translate to machine code.
- Why is it not this easy?

Forward References

```plaintext
mov ax, 0
cmp ax, bx
jge next ; can’t assemble
....
next:
```

Some Solutions

1. When a forward ref. is found:
   - put the statement in a table
   - at the end of the pass, assemble the statements in the table
   - disadvantage – complex – pieces must be put back in the right order.
2. More common: two pass assembly
   - pass 1 – build symbol table
   - pass 2 – assemble code
Pass 1

- Builds the symbol table, uses an opcode table.
- Symbol table – one entry for each user-defined symbol in the program. Symbols defined by equates or used as labels.

Symbol Table Example

- Symbol Table Example

AddNumber EQU 12h
ArraySize   EQU 3h
.data
TestArray   DW 1, 2, 3
.code
Start: MOV AX, offset TestArray
Next: PUSH AX
       MOV AX, AddNumber
       PUSH AX
       ...

Symbol Table, cont.

- To build the symbol table, the assembler uses the **location counter** and **opcode table**.
- Location counter – set to zero at the beginning, increased by the instruction length for each instruction processed.
- Opcode table – used to look up length of each instruction.

OpCode Table

- Instruction length – used to update the location counter in pass 1
- Instruction class – sends the assembler off to a routine that process all similar instructions (all reg-to-reg for example)
Pass 2

• Assembler goes through program again, using the symbol table and opcode table to generate machine code.
  – LEA AX, TestArray

LEA – looks up opcode in opcode table
TestArray – plugs in the value from the symbol table along with any relocation information.
• Assembles instruction, places in output buffer.

Linking and Loading

• Large programs are developed as independently-assembled modules.
• Problems addressed by linker:
  – relocation problem
  – external reference problem

Separately Assembled Files

• Example from old version of Tanenbaum (figure 7-13, 7-14)

• Basic problem – B is referred to in one module, defined in another. We need to hook them together.
• Points to note:
  – only one module should have .startup and .exit
  – only one module should set up the stack (both modules will share the same stack)
Global Label Definition

• Assembler destroys the symbol table after assembly – assumed scope of labels is local.
• To declare them as global:
  PUBLIC – use this when they are defined; makes the symbol global and keeps the definition for the linker.
  EXTRN – use this when label is referenced.

EXTRN formats

• EXTRN Name:Type
  – Type:
    • Byte, Word, Dword for data
    • Near, Far for procedures
    • or Proc, which defaults to Near for a small memory model.

• example (on board)

• external procedure call example, Irvine P. 335
<table>
<thead>
<tr>
<th>High Level Language Interface</th>
<th>Naming Convention</th>
</tr>
</thead>
</table>
| Frequently only parts of an application are written in assembly language:  
  - parts that need be optimized for speed  
  - parts that need to communicate more directly with the hardware  
| Must understand HLL’s:  
  - naming convention  
  - memory model  
  - calling conventions | C pre-pends an underscore to external identifiers:  
  extern int addem(int num1, int num2)  
  ...  
  total = addem(5,6)  

- in assembly language subroutine, define:  
  PUBLIC _addem  
- C expects the case to be the same in both modules. |

<table>
<thead>
<tr>
<th>Memory Model</th>
<th>Calling Conventions</th>
</tr>
</thead>
</table>
| small is used by default but it can be changed. | C passes parameters in reverse order:  
  total = addem(X, Y)  
  Y is pushed on the stack first, then X.  
- C expects function results in a register:  
  - 16-bit result in AX  
  - 32-bit result in DX:AX  
  Parameters can be passed by value or reference (& in parameter list means to put the address of the parameter on the stack). |

.model small
Returning from ASM to C

• C generates code to clean up the stack (remove the parameters).
• The ASM module should use RET with no argument.

/* This C program computes \((A^2 + B^2) / (C^2)\) by calling
the separately-assembled routine SQUARIT.
C pushes the parameters on the stack in reverse order. */

#include <stdio.h>

main()
{
    extern void SQUARIT(int, int, int, int*);
    int a=5;
    int b=3;
    int c=2;
    int ans;
    SQUARIT (a, b, c, &ans);
    printf("The answer is: %d\n", ans);
}

Interfacing with Visual C++

• You’ll get to try it in lab!
• Biggest difference: Visual C++ generates 32-bit applications.
/* This C program computes \((A^2 + B^2) / (C^2)\) by calling the separately-assembled routine SQUARIT. C pushes the parameters on the stack in reverse order. */

#include <stdio.h>
extern "C" int SQUARIT(int, int, int *);

main()
{
    int a=5;
    int b=3;
    int c=2;
    int ans;

    SQUARIT(a, b, c, &ans);
    printf("The answer is: %d\n", ans);
}

PUBLIC SQUARIT
model small
.code
    .166
    push ebp           ; initialize reference point to
    mov ebp, esp      ; this stack frame
    push esp
    push ebx
    push eax
    mov eax, [ebp-8]  ; move A into AX
    imul eax, [ebp-8] ; store A*A as a local variable
    mov [ebp-8], eax  ; store B*B locally
    mov eax, [ebp-12] ; move B into AX
    imul eax, [ebp-12] ; store B*B locally
    mov [ebp-16], eax
    mov eax, [ebp-16] ; store C*C locally
    add eax, [ebp-20] ; get A*A + B*B
    sub eax, [ebp-20] ; subtract C*C (answer in AX)
    mov ebx, eax      ; store result at ans
    pop ebx
    pop eax
    pop esp
    ret               ; calling program responsible for adding 16 to esp