Lecture 18

- Assembly
- Linking and Loading
- Assembling Separate Files (review)
- C++/Assembly Interface (needed for Lab 5!)

Assembly Process (Review)

- Assembler translates symbolic assembly language program into numeric machine code.

Forward References

```assembly
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?

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

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.

- 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 processes 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 – assembler looks up opcode in opcode table
  TestArray – assembler 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 (review)

- 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.

High Level Language Interface

• external procedure call example, Irvine P. 335

• Frequently only parts of an application are written in assembly language:

  • Must understand HLL’s:
Naming Convention

- C pre-pends an underscore to external identifiers:
  ```c
  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.

Memory Model

- small is used by default but it can be changed.
  ```
  .model small
  ```

Calling Conventions

- C passes parameters in reverse order:

- C expects function results in a register:

- Types of parameters?

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);
}

/** 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, 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);
}

Interfacing with Visual C++

• You’ll get to try it in lab!
• Biggest difference: Visual C++ generates 32-bit applications.
PUBLIC_SQUARIT
.model small
.code

_SQUARIT:
push ebp        ;initialize reference point to
mov ebp, esp    ;top stack frame
push eax        ;save registers used by this procedure
push edx
push esi

mov eax, [ebp-8]           ;mov A into AX
imul dword ptr [ebp+8]    ;A*A
mov [ebp-16], eax          ;store A*A as a local variable
mov eax, [ebp-12]          ;mov B into AX
imul dword ptr [ebp+12]   ;B*B
mov [ebp-20], eax          ;store B*B locally
mov eax, [ebp-16]          ;mov C
imul dword ptr [ebp+16]   ;C*C
mov [ebp-24], eax          ;store C*C locally
add eax, [ebp-16]         ;get A*A
cdq
mov eax, [ebp-20]          ;A*A + B*B in EDX:EAX
mov edx, [ebp-24]          ;divide by C*C (answer in AX)
mov edi, [ebp-20]          ;put address of ans in EDI
mov [edi], eax             ;store result at ans
pop edi                   ;restore registers
pop edx
pop eax
pop ebp
ret                        ;calling program responsible
end                        ;for adding 16 to esp