Advanced Procedures

Computer Organization and Assembly Languages

Yung-Yu Chuang

2005/11/24

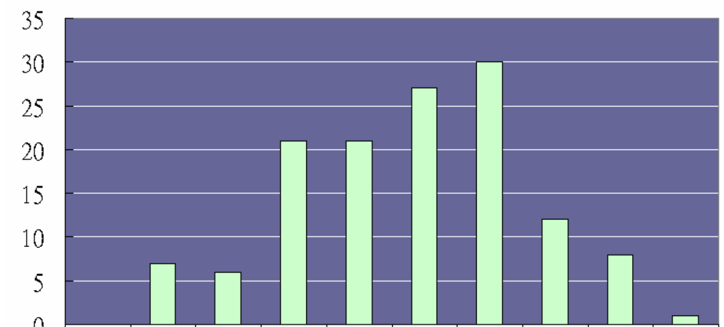
with slides by Kip Irvine

Announcements

- Assignment #2 due today by midnight.
- Assignment #3 due on 12/6 (Tue) midnight.
- Graded midterm will be returned later today.

Statistics for midterm

- Average=52.82, Std. dev.=18.01



0~9	10~19	20~29	30~39	40~49	50~59	60~69	70~79	80~89	90~99
0	7	6	21	21	27	30	12	8	1

Overview

- Local Variables (creating and initializing on the stack, scope and lifetime, LOCAL)
- Stack Parameters (INVOKE, PROC, PROTO, passing by value, passing by reference, memory model and language specifiers)
- Stack Frames (access by indirect addressing)
- Recursion
- Creating Multimodule Programs

Local variables

- The variables defined in the data segment can be taken as *static global variables*.

visibility=the whole program

lifetime=program duration

- A local variable is created, used, and destroyed within a single procedure
- Advantages of local variables:
  - Restricted access: easy to debug, less error prone
  - Efficient memory usage
  - Same names can be used in two different procedures

Local variables

- The LOCAL directive declares a list of local variables
  - immediately follows the PROC directive
  - each variable is assigned a type
- Syntax:

  LOCAL *varlist*

Example:

```
MySub PROC
   LOCAL var1:BYTE, var2:WORD, var3:SDWORD
```

Local variables

Examples:

```
LOCAL flagVals[20]:BYTE  ; array of bytes

LOCAL pArray:PTR WORD  ; pointer to an array

myProc PROC,         ; procedure
   p1:PTR WORD       ; parameter
   LOCAL t1:BYTE,    ; local variables
   t2:WORD,
   t3:DWORD,
   t4:PTR DWORD
```

MASM-generated code

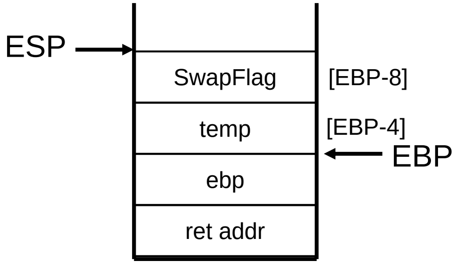
```
BubbleSort PROC
   LOCAL temp:DWORD, SwapFlag:BYTE
   . . .
   ret
BubbleSort ENDP
```

MASM generates the following code:

```
BubbleSort PROC
   push ebp
   mov  ebp,esp
   add  esp,0FFFFFFF8h ; add -8 to ESP
   . . .
   mov  esp,ebp
   pop  ebp
   ret
BubbleSort ENDP
```

MASM-generated code

Diagram of the stack frame for the BubbleSort procedure:



Reserving stack space

- .stack 4096
- **Sub1** calls **Sub2**, **Sub2** calls **Sub3**

```
Sub1 PROC
  LOCAL array1[50]:DWORD ; 200 bytes

Sub2 PROC
  LOCAL array2[80]:WORD  ; 160 bytes

Sub3 PROC
  LOCAL array3[300]:WORD ; 300 bytes
```

Register vs. stack parameters

- Register parameters require dedicating a register to each parameter. Stack parameters are more convenient
- Imagine two possible ways of calling the DumpMem procedure. Clearly the second is easier:

```
pushad
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
popad
```

```
push OFFSET array
push LENGTHOF array
push TYPE array
call DumpMem
```

INVOKE directive

- The INVOKE directive is a powerful replacement for Intel's CALL instruction that lets you pass multiple arguments
- Syntax:
  INVOKE *procedureName* [, *argumentList*]
- *ArgumentList* is an optional comma-delimited list of procedure arguments
- Arguments can be:
  - immediate values and integer expressions
  - variable names
  - address and ADDR expressions
  - register names

INVOKE examples

```
.data
byteVal BYTE 10
wordVal WORD 1000h
.code
   ; direct operands:
   INVOKE Sub1,byteVal,wordVal

   ; address of variable:
   INVOKE Sub2,ADDR byteVal

   ; register name, integer expression:
   INVOKE Sub3,eax,(10 * 20)

   ; address expression (indirect operand):
   INVOKE Sub4,[ebx]
```

INVOKE example

```
.data
val1 DWORD 12345h
val2 DWORD 23456h
.code
  INVOKE AddTwo, val1, val2

push val1
push val2
call AddTwo
```

ADDR operator

- Returns a near or far pointer to a variable, depending on which memory model your program uses:
  - Small model: returns 16-bit offset
  - Large model: returns 32-bit segment/offset
  - Flat model: returns 32-bit offset
- Simple example:

```
.data
myWord WORD ?
.code
INVOKE mySub,ADDR myWord
```

PROC directive

- The **PROC** directive declares a procedure with an optional list of named parameters.
- Syntax:

  label **PROC paramList**
- ***paramList*** is a list of parameters separated by commas. Each parameter has the following syntax:

  paramName:*type*

type must either be one of the standard ASM types (BYTE, SBYTE, WORD, etc.), or it can be a pointer to one of these types.

PROC examples

- The AddTwo procedure receives two integers and returns their sum in EAX.
- C++ programs typically return 32-bit integers from functions in EAX.

```
AddTwo PROC,
   val1:DWORD, val2:DWORD

   mov eax,val1
   add eax,val2
   ret
AddTwo ENDP
```

PROC examples

FillArray receives a pointer to an array of bytes, a single byte fill value that will be copied to each element of the array, and the size of the array.

```
FillArray PROC,
   pArray:PTR BYTE, fillVal:BYTE
   arraySize:DWORD

   mov ecx,arraySize
   mov esi,pArray
   mov al,fillVal
L1:mov [esi],al
   inc esi
   loop L1
   ret
FillArray ENDP
```

PROTO directive

- Creates a procedure prototype
- Syntax:
  - ***label*** **PROTO** ***paramList***
- Every procedure called by the **INVOKE** directive must have a prototype
- A complete procedure definition can also serve as its own prototype

PROTO directive

- Standard configuration: PROTO appears at top of the program listing, INVOKE appears in the code segment, and the procedure implementation occurs later in the program:

```
MySub PROTO    ; procedure prototype

.code
INVOKE MySub   ; procedure call


MySub PROC     ; procedure implementation
   .
   .
MySub ENDP
```

PROTO example

- Prototype for the ArraySum procedure, showing its parameter list:

```
ArraySum PROTO,
    ptrArray:PTR DWORD,  ; points to the array
    szArray:DWORD        ; array size
```

Passing by value

- When a procedure argument is passed by value, a copy of a 16-bit or 32-bit integer is pushed on the stack. Example:

```
.data
myData WORD 1000h
.code
main PROC
    INVOKE Sub1, myData
```

MASM generates the following code:

```
push myData
call Sub1
```

Passing by reference

- When an argument is passed by reference, its address is pushed on the stack. Example:

```
.data
myData WORD 1000h
.code
main PROC
    INVOKE Sub1, ADDR myData
```

MASM generates the following code:

```
push OFFSET myData
call Sub1
```

Parameter classifications

- An input parameter is data passed by a calling program to a procedure.
  - The called procedure is not expected to modify the corresponding parameter variable, and even if it does, the modification is confined to the procedure itself.
- An output parameter is created by passing a pointer to a variable when a procedure is called.
  - The procedure does not use any existing data from the variable, but it fills in a new value before it returns.
- An input-output parameter represents a value passed as input to a procedure, which the procedure may modify.
  - The same parameter is then able to return the changed data to the calling program.

Example: exchanging two integers

The Swap procedure exchanges the values of two 32-bit integers. pValX and pValY do not change values, but the integers they point to are modified.

```
Swap PROC USES eax esi edi,
   pValX:PTR DWORD, ; pointer to first integer
   pValY:PTR DWORD  ; pointer to second integer

   mov esi,pValX    ; get pointers
   mov edi,pValY
   mov eax,[esi]    ; get first integer
   xchg eax,[edi]   ; exchange with second
   mov [esi],eax    ; replace first integer
   ret
Swap ENDP
```

Stack frame

- Also known as an activation record
- Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
- Created by the following steps:
  - Calling program pushes arguments on the stack and calls the procedure.
  - The called procedure pushes EBP on the stack, and sets EBP to ESP.
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.

Memory models

- A program's memory model determines the number and sizes of code and data segments.
- Real-address mode supports tiny, small, medium, compact, large, and huge models.
- Protected mode supports only the flat model.

Small model: code < 64 KB, data (including stack) < 64 KB. All offsets are 16 bits.

Flat model: single segment for code and data, up to 4 GB. All offsets are 32 bits.

`.MODEL` directive

- **`.MODEL`** directive specifies a program's memory model and model options (language-specifier).
- Syntax:

  .MODEL *memorymodel* [,*modeloptions*]
- ***memorymodel*** can be one of the following:
  - tiny, small, medium, compact, large, huge, or flat
- ***modeloptions*** includes the language specifier:
  - procedure naming scheme
  - parameter passing conventions

Language specifiers

- C:
  - procedure arguments pushed on stack in reverse order (right to left)
  - calling program cleans up the stack
- pascal
  - procedure arguments pushed in forward order (left to right)
  - called procedure cleans up the stack
- stdcall
  - procedure arguments pushed on stack in reverse order (right to left)
  - called procedure cleans up the stack

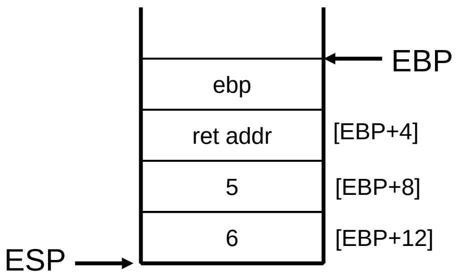
Explicit access to stack parameters

- A procedure can explicitly access stack parameters using constant offsets from EBP.
  - Example: [ebp + 8]
- EBP is often called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the procedure.
- EBP must be restored to its original value when a procedure returns.

Stack frame example

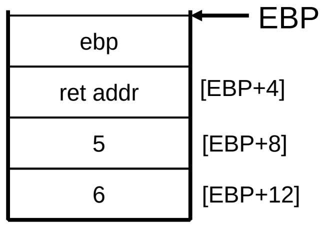
```
.data
sum DWORD ?
.code
  push 6            ; second argument
  push 5            ; first argument
  call AddTwo       ; EAX = sum
  mov  sum,eax      ; save the sum
```

```
AddTwo PROC
  push ebp
  mov  ebp,esp
  .
  .
```



Stack frame example

```
AddTwo PROC
  push ebp
  mov eax,esp            ; base of stack frame
  mov eax,[ebp + 12]     ; second argument (6)
  add eax,[ebp + 8]      ; first argument (5)
  pop ebp
  ret 8                  ; clean up the stack
AddTwo ENDP             ; EAX contains the sum
```



Passing arguments by reference

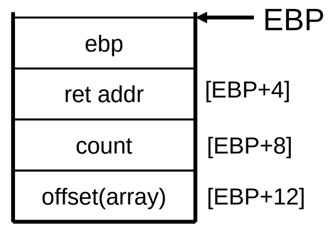
- The ArrayFill procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

```
.data
count = 100
array WORD count DUP(?)
.code
   push OFFSET array
   push COUNT
   call ArrayFill
```

Passing arguments by reference

`ArrayFill` can reference an array without knowing the array's name:

```
ArrayFill PROC
   push ebp
   mov  ebp,esp
   pushad
   mov  esi,[ebp+12]
   mov  ecx,[ebp+8]
   .
   .
```



LEA instruction (load effective address)

- The LEA instruction returns offsets of both direct and indirect operands.
  - OFFSET operator can only return constant offsets.
- LEA is required when obtaining the offset of a stack parameter or local variable. For example:

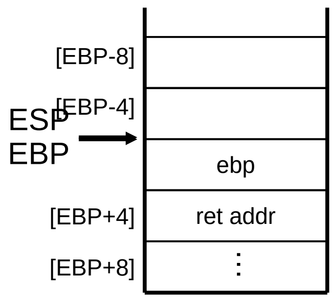
```
CopyString PROC,
   count:DWORD
   LOCAL temp[20]:BYTE

   mov edi,OFFSET count; invalid operand
   mov esi,OFFSET temp ; invalid operand
   lea edi,count       ; ok
   lea esi,temp        ; ok
```

Creating local variables

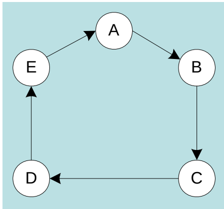
- To explicitly create local variables, subtract their total size from ESP.
- The following example creates and initializes two 32-bit local variables (we'll call them locA and locB):

```
MySub PROC
   push ebp
   mov  ebp,esp
   sub  esp,8
   mov  [ebp-4],123456h
   mov  [ebp-8],0
   .
   .
```



Recursion

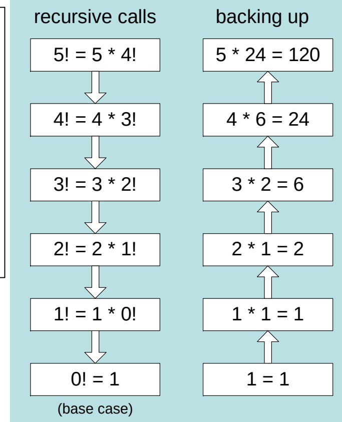
- The process created when . . .
  - A procedure calls itself
  - Procedure A calls procedure B, which in turn calls procedure A
- Using a graph in which each node is a procedure and each edge is a procedure call, recursion forms a cycle:



Calculating a factorial

This function calculates the factorial of integer *n*. A new value of *n* is saved in each stack frame:

```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n*factorial(n-1);
}
```




```
factorial(5);
```

Calculating a factorial

```
Factorial PROC
   push ebp
   mov  ebp,esp
   mov  eax,[ebp+8]    ; get n
   cmp  eax,0          ; n < 0?
   ja   L1             ; yes: continue
   mov  eax,1          ; no: return 1
   jmp  L2
L1:dec  eax
   push eax            ; Factorial(n-1)
   call Factorial

ReturnFact:
   mov  ebx,[ebp+8]      ; get n
   mul  ebx              ; ax = ax * bx

L2:pop  ebp            ; return EAX
   ret  4              ; clean up stack
Factorial ENDP
```

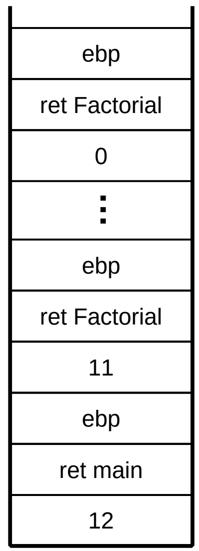
Calculating a factorial

```
push 12
call Factorial
```

```
Factorial PROC
   push ebp
   mov  ebp,esp
   mov  eax,[ebp+8]
   cmp  eax,0
   ja   L1
   mov  eax,1
   jmp  L2
L1:dec  eax
   push eax
   call Factorial

ReturnFact:
   mov  ebx,[ebp+8]
   mul  ebx

L2:pop  ebp
   ret  4
Factorial ENDP
```



Multimodule programs

- A multimodule program is a program whose source code has been divided up into separate ASM files.
- Each ASM file (module) is assembled into a separate OBJ file.
- All OBJ files belonging to the same program are linked using the link utility into a single EXE file.
  - This process is called static linking

Advantages

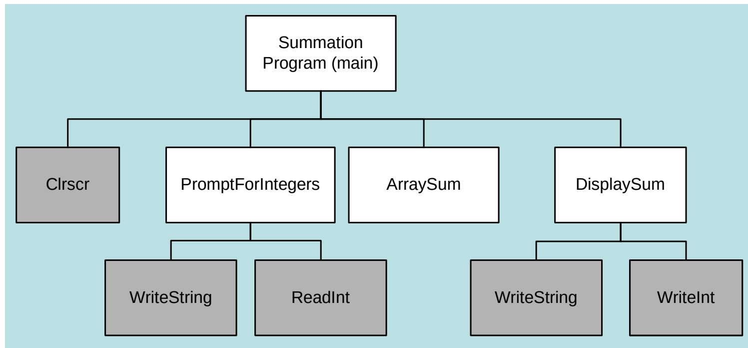
- Large programs are easier to write, maintain, and debug when divided into separate source code modules.
  - When changing a line of code, only its enclosing module needs to be assembled again. Linking assembled modules requires little time.
  - A module can be a container for logically related code and data
    - encapsulation: procedures and variables are automatically hidden in a module unless you declare them public

Creating a multimodule program

- Here are some basic steps to follow when creating a multimodule program:
  - Create the main module
  - Create a separate source code module for each procedure or set of related procedures
  - Create an include file that contains procedure prototypes for external procedures (ones that are called between modules)
  - Use the INCLUDE directive to make your procedure prototypes available to each module

Example: ArraySum Program

- Let's review the ArraySum program from Chapter 5.



Each of the four white rectangles will become a module.

INCLUDE file

The sum.inc file contains prototypes for external functions that are not in the Irvine32 library:

```
INCLUDE Irvine32.inc

PromptForIntegers PROTO,
   ptrPrompt:PTR BYTE,      ; prompt string
   ptrArray:PTR DWORD,      ; points to the array
   arraySize:DWORD          ; size of the array

ArraySum PROTO,
   ptrArray:PTR DWORD,      ; points to the array
   count:DWORD              ; size of the array

DisplaySum PROTO,
   ptrPrompt:PTR BYTE,      ; prompt string
   theSum:DWORD             ; sum of the array
```

Main.asm

```
TITLE Integer Summation Program

INCLUDE sum.inc

.code
main PROC
   call Clrscr

   INVOKE PromptForIntegers,
      ADDR prompt1,
      ADDR array,
      Count

      ...
      call Crlf
      INVOKE ExitProcess,0
main ENDP
END main
```