Computer Organization & Assembly Language - EE2003

Lecture 13

Week 08

Chapter Overview

- Stack Frames
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Java Bytecodes

Stack Frames

- Stack Parameters
- Local Variables
- ENTER and LEAVE Instructions
- LOCAL Directive
- WriteStackFrame Procedure

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.

Stack Parameters

- More convenient than register parameters
- Two possible ways of calling DumpMem. Which is easier?

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

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

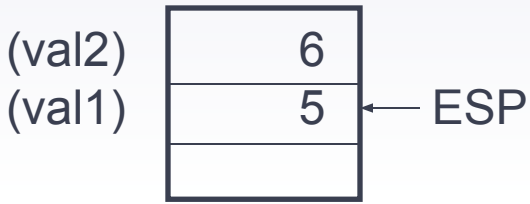
Passing Arguments by Value

- Push argument values on stack
  - (Use only 32-bit values in protected mode to keep the stack aligned)
- Call the called-procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called-procedure did not remove them

Example

```
.data
val1   DWORD 5
val2   DWORD 6

.code
push val2
push val1
```



Stack prior to CALL

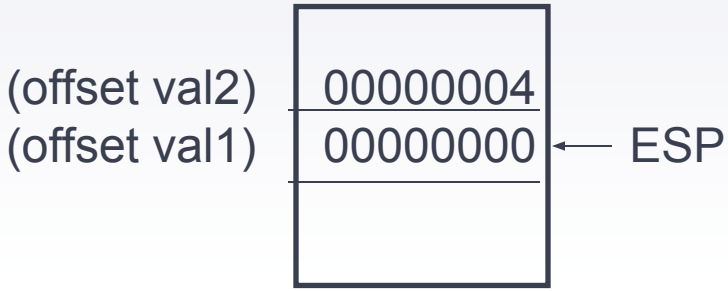
Passing by Reference

- Push the offsets of arguments on the stack
- Call the procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called procedure did not remove them

Example

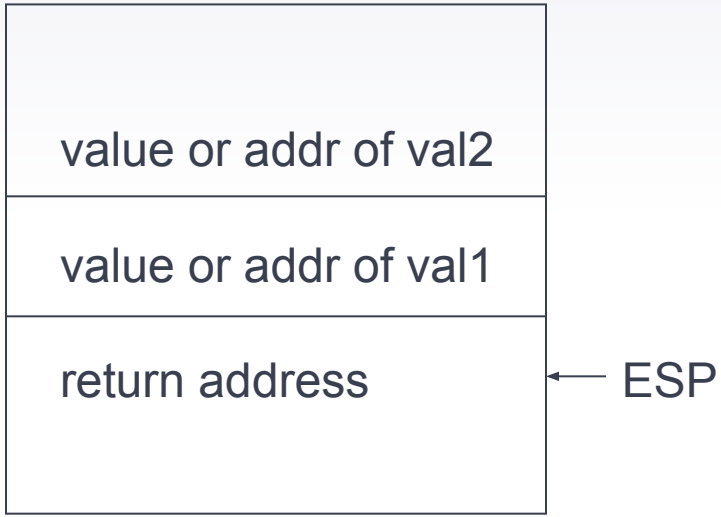
```
.data
val1   DWORD 5
val2   DWORD 6

.code
push OFFSET val2
push OFFSET val1
```



Stack prior to CALL

Stack after the CALL



Passing an Array by Reference (1 of 2)

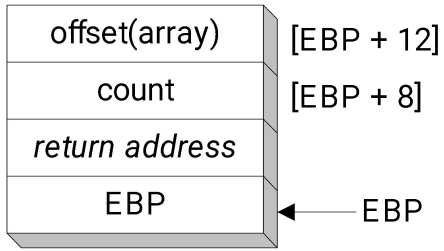
- The ArrayFill procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

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

Passing an Array by Reference (2 of 2)

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]
     .
     .
```



ESI points to the beginning of the array, so it's easy to use a loop to access each array element. View the complete program.

Accessing Stack Parameters (C/C++)

- C and C++ functions access stack parameters using constant offsets from EBP[1].
  - Example: [ebp + 8]
- EBP is called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the function.
- EBP must be restored to its original value when a function returns.

[1] BP in Real-address mode

RET Instruction

- *Return from subroutine*
- Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
- Syntax:
  - RET
  - RET *n*
- Optional operand *n* causes *n* bytes to be added to the stack pointer after EIP (or IP) is assigned a value.

Who removes parameters from the stack?

Caller (C) ..... or ..... Called-procedure (STDCALL):

```
                                AddTwo PROC
push val2               push  ebp
push val1               mov   ebp,esp
call AddTwo             mov   eax,[ebp+12]
add  esp,8              add   eax,[ebp+8]

                        pop   ebp
                        ret   8
```

(Covered later: The MODEL directive specifies calling conventions)

Your turn . . .

- Create a procedure named Difference that subtracts the first argument from the second one. Following is a sample call:
  - push 14 ; first argument
  - push 30 ; second argument
  - call Difference; EAX = 16

```
Difference PROC
push ebp
mov  ebp,esp
mov  eax,[ebp + 8]     ; second argument
sub  eax,[ebp + 12]    ; first argument
pop  ebp
ret  8
Difference ENDP
```

Passing 8-bit and 16-bit Arguments

- Cannot push 8-bit values on stack
- Pushing 16-bit operand may cause page fault or ESP alignment problem
  - incompatible with Windows API functions
- Expand smaller arguments into 32-bit values, using MOVZX or MOVSX:

```
.data
charVal BYTE 'x'
.code
        movzx     eax,charVal
        push eax
        call  Uppercase
```

Passing Multiword Arguments

- Push high-order values on the stack first; work backward in memory
- Results in little-endian ordering of data
- Example:
  - .data
  - longVal DQ 1234567800ABCDEFh
  - .code
  - push DWORD PTR longVal + 4 ; high doubleword
  - push DWORD PTR longVal ; low doubleword
  - call WriteHex64

Saving and Restoring Registers

- Push registers on stack just after assigning ESP to EBP
  - local registers are modified inside the procedure

```
MySub PROC
        push    ebp
        movebp,esp
        push    ecx         ; save local registers
        push    edx
```

Stack Affected by USES Operator

- MySub1 PROC USES ecx edx
- ret
- MySub1 ENDP

- USES operator generates code to save and restore registers:

  - MySub1 PROC
  - push ecx
  - push edx

  - pop edx

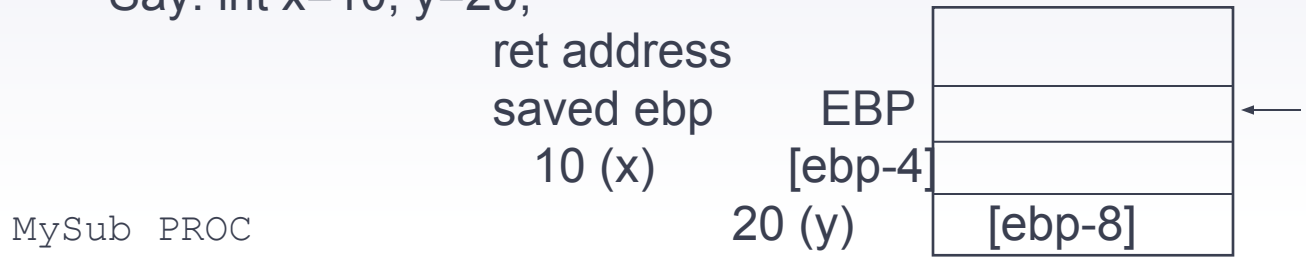
Local Variables

- Only statements within subroutine can view or modify local variables
- Storage used by local variables is released when subroutine ends
- local variable name can have the same name as a local variable in another function without creating a name clash
- Essential when writing recursive procedures, as well as procedures executed by multiple execution threads

Creating LOCAL Variables

Example - create two DWORD local variables:

Say: int x=10, y=20;




```
MySub PROC
    push    ebp
    mov ebp,esp
    sub   esp,8     ;create 2 DWORD variables

    mov DWORD PTR [ebp-4],10 ; initialize x=10
    mov DWORD PTR [ebp-8],20 ; initialize y=20
```

LEA Instruction

- LEA returns offsets of direct and indirect operands
  - OFFSET operator only returns constant offsets
- LEA required when obtaining offsets of stack parameters & local variables
- 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
```

LEA Example

Suppose you have a Local variable at [ebp-8]

And you need the address of that local variable in ESI

You cannot use this:

```
mov esi, OFFSET [ebp-8]      ; error
```

Use this instead:

```
lea esi,[ebp-8]
```

ENTER Instruction

- ENTER instruction creates stack frame for a called procedure
  - pushes EBP on the stack
  - sets EBP to the base of the stack frame
  - reserves space for local variables
  - Example:
    - MySub PROC
    - enter 8,0
  - Equivalent to:
    - MySub PROC
    - push ebp
    - mov ebp,esp

LEAVE Instruction

Terminates the stack frame for a procedure.

```
MySub PROC
        enter 8,0
        ...
        ...
        ...
        leave
        ret
MySub ENDP
```

Equivalent operations

```
push    ebp
mov ebp,esp
sub  esp,8      ; 2 local DWORDs
```

```
mov esp,ebp  ; free local space
pop ebp
```

LOCAL Directive

- 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
```

Using LOCAL

Examples:

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

LOCAL pArray:PTR WORD ; pointer to an array

myProc PROC,  ; procedure
     LOCAL t1:BYTE,    ; local variables
```

LOCAL Example (1 of 2)

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

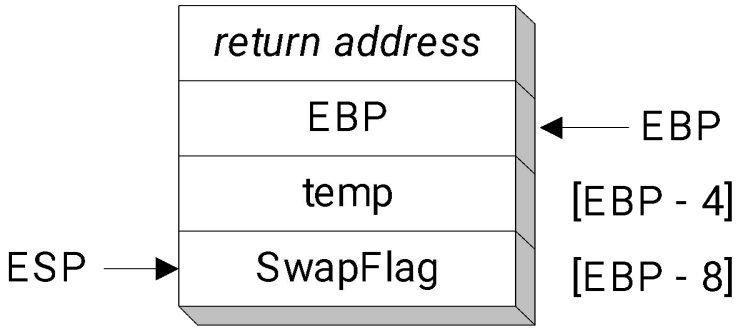
MASM generates the following code:

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

]

LOCAL Example (2 of 2)

Diagram of the stack frame for the BubbleSort procedure:



What's Next

- Stack Frames
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Java Bytecodes

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- ADDR Operator
- PROC Directive
- PROTO Directive
- Parameter Classifications
- Example: Exchaning Two Integers
- Debugging Tips

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]
```

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 (1 of 2)

- 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 Directive (2 of 2)

- Alternate format permits parameter list to be on one or more separate lines:

label PROC,

paramList ← comma required

- The parameters can be on the same line . . .

param-1:type-1, param-2:type-2, . . ., param-n:type-n

- Or they can be on separate lines:

param-1:type-1,

param-2:type-2,

. . .,

param-n:type-n

AddTwo Procedure (1 of 2)

- The AddTwo procedure receives two integers and returns their sum in EAX.

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

     mov eax,val1
     add eax,val2

     ret
AddTwo ENDP
```

PROC Examples (2 of 3)

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
```

PROC Examples (3 of 3)

```
Swap PROC,
pValX:PTR DWORD,
pValY:PTR DWORD
. . .
Swap ENDP
```

```
ReadFile PROC,
     pBuffer:PTR BYTE
     LOCAL fileHandle:DWORD
     . . .
ReadFile 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
```

THANKS!

Any questions?

You can find me at:

- A.qadeer@nu.edu.pk
- Office #213, Visiting Hours Only