Chapter Overview

• **Stack Frames**
• Recursion
• .MODEL Directive
• INVOKE, ADDR, PROC, and PROTO
• Creating Multimodule Programs
Stack Frames

• Stack Parameters
• Local Variables
• ENTER and LEAVE Instructions
• LOCAL Directive
• WriteStackFrame Procedure
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
```
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
  - Call the procedure, and cause the return address to be pushed on the stack
  - The called procedure pushes EBP on the stack, and sets EBP to ESP. (EBP acts as base reference now)
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.
  - If any registers need to be saved, they are pushed on the stack
Explicit Access to Stack Parameters

• A procedure can explicitly access stack parameters using constant offsets from EBP\(^1\).
  – 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.

\(^1\) 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.
  – It is used to clean up the stack
  – The *n* must equal the number of bytes of stack space consumed by the subroutine parameters
Stack Frame Example (1 of 2)

.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
  .
  .
  .
  .
  .
  .
  .
  .
  .
  .
  .
  .[
  00000006
  [EBP + 12]
  00000005
  [EBP + 8]
  return address
  .
  [EBP + 4]
  EBP
  .
  .
  .
  EBP, ESP

Web site  Examples
AddTwo Procedure (1 of 2)

• Recall the AddTwo Procedure

    AddTwo PROC,
    val1:DWORD, val2:DWORD

    mov eax, val1
    add eax, val2
    ret
    AddTwo ENDP
AddTwo Procedure (2 of 2)

- MASM generates the following code when we assemble AddTwo (from the previous panel):

```assembly
AddTwo PROC,
    val1:DWORD, val2:DWORD
    push ebp
    mov ebp, esp ; frame pointer (ebp)
    mov eax,val1
    add eax,val2
    mov esp, ebp ; point to stack pointer (esp)
    pop ebp
    mov esp, ebp ; remove local variables from stack
    ret 8 ; 8 means to clean up stack (remove passed parameters of 8-bytes from stack)
AddTwo ENDP
```
Passing Arguments 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:

```assembly
.data
count = 100
array WORD count DUP(?)
.code
    push OFFSET array
    push COUNT
    call ArrayFill
```
Passing Arguments 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.
```
Local Variables

- To create local variables, subtract their total size from ESP.
- Example for two 32-bit local variables (locA and locB)

```assembly
MySub PROC
    push ebp
    mov ebp, esp
    sub esp, 8 ; reserve space in stack
    mov [ebp-4], 12h ; locA
    mov [ebp-8], 34h ; locB
    mov esp, ebp ; remove locals
    pop ebp
    ret ; no passed parameters to clean up
MySub PROC
```

Examples

LEA Instruction

• 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
```

Irvine, Kip R. Assembly Language for Intel-Based Computers 5/e, 2007. 14
Your turn . . .

• Create a procedure named **Difference** that subtracts the first argument from the second one. Following is a sample call:

```assembly
push 14 ; first argument
push 30 ; second argument
call Difference ; EAX = 16
```

```assembly
Difference PROC
    push ebp
    mov ebp, esp
    mov eax, [ebp + 8] ; second argument
    sub eax, [ebp + 12] ; first argument
    pop ebp
    ret 8 ; clean up arguments
Difference ENDP
```
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** is a pointer to a variable containing input that will be both used and modified by the procedure.
  - The variable passed by the calling program is modified.
Example: Exchanging Two Integers

The Swap procedure exchanges the values of two 32-bit integers. \texttt{pValX} and \texttt{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
```
ENTER and LEAVE

• 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 1: no locals

MySub PROC
enter 0,0

Equivalent to:

MySub PROC
push ebp
mov ebp, esp

Example 2: locals of 8 bytes

MySub PROC
enter 8,0

Equivalent to:

MySub PROC
push ebp
mov ebp, esp
sub esp, 8

LOCAL Directive

• A local variable is created, used, and destroyed within a single procedure
• 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:DWORD
```
Using LOCAL

Examples:

LOCAL flagVals[20]:BYTE ; array of bytes

LOCAL pArray:PTR WORD ; pointer to an array

myProc PROC,
  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:

```
return address

EBP

temp

SwapFlag

ESP
```
Non-Doubleword Local Variables

• Local variables can be different sizes
• How created in the stack by LOCAL directive:
  – 8-bit: assigned to next available byte
  – 16-bit: assigned to next even (word) boundary
  – 32-bit: assigned to next doubleword boundary
Local Byte Variable

Example1 PROC
   LOCAL var1:BYTE
   mov al,var1 ; [EBP - 1]
   ret
Example1 ENDP
WriteStackFrame Procedure

• Displays contents of current stack frame
  – Prototype:

    WriteStackFrame PROTO,
    numParam:DWORD,    ; # of passed parameters
    numLocalVal: DWORD, ; # of DWordLocal variables
    numSavedReg: DWORD  ; # of saved registers
WriteStackFrame Example

main PROC
  mov eax, 0EAEAEAEAh
  mov ebx, 0EBEBEBEBBh
  INVOKE aProc, 1111h, 2222h
  exit
main ENDP

aProc PROC USES eax ebx,
  x: DWORD, y: DWORD
LOCAL a:DWORD, b:DWORD
PARAMS = 2
LOCALS = 2
SAVED_REGS = 2
mov a,0AAAAh
mov b,0BBBBh
INVOKE WriteStackFrame, PARAMS, LOCALS, SAVED_REGS

Output

Stack frame

00002222 ebp+12 (param)
00001111 ebp+8 (param)
00401083 ebp+4
0012FFF0 ebp+0 ← ebp
0000AAAA ebp−4 (local)
0000BBBB ebp−8 (local)
EAEAEAEAE ebp−12 (save reg)
EBEBEBEBB ebp−16 (save reg)
What's Next

- Stack Frames
- **Recursion**
- .MODEL Directive
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
Recursion

• What is recursion?
• Recursively Calculating a Sum
• Calculating a Factorial
What is 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:
Recursively Calculating a Sum

The CalcSum procedure recursively calculates the sum of an array of integers. Receives: ECX = count. Returns: EAX = sum

CalcSum PROC
    cmp ecx, 0 ; check counter value
    jz L2 ; quit if zero
    add eax, ecx ; otherwise, add to sum
    dec ecx ; decrement counter
    call CalcSum ; recursive call
L2: ret
CalcSum ENDP

Stack frame:

<table>
<thead>
<tr>
<th>Pushed On Stack</th>
<th>ECX</th>
<th>EAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>L2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>L2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>L2</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>L2</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>L2</td>
<td>0</td>
<td>15</td>
</tr>
</tbody>
</table>

View the complete program
Calculating a Factorial  (1 of 3)

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

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

As each call instance returns, the product it returns is multiplied by the previous value of $n$. 

![Diagram showing recursive calls and backing up](image-url)
Calculating a Factorial (2 of 3)

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

; Instructions from this point on execute when each
; recursive call returns.

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

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

See the program listing
Calculating a Factorial  (3 of 3)

Suppose we want to calculate 12!

This diagram shows the first few stack frames created by recursive calls to Factorial

Each recursive call uses 12 bytes of stack space.
What's Next

• Stack Frames
• Recursion
• .MODEL Directive
• INVOKE, ADDR, PROC, and PROTO
• Creating Multimodule Programs
.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
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.
Language Specifiers

• **C:**
  – procedure arguments pushed on stack in reverse order (right to left)
  – calling program cleans up the stack

• **STDCALL**
  – procedure arguments pushed on stack in reverse order (right to left)
  – called procedure cleans up the stack
e.g., ret 8  ; add 8 to ESP to clean up stack
Review Questions

1. Describe the small memory model.
2. Describe the flat memory model.
3. How is the C language option (of the .MODEL directive) different from that of STDCALL in regard to removing arguments from the stack?
What's Next

• Stack Frames
• Recursion
• .MODEL Directive
• **INVOKE, ADDR, PROC, and PROTO**
• Creating Multimodule Programs
INVOKE, ADDR, PROC, and PROTO

• INVOKE Directive
• ADDR Operator
• PROC Directive
• PROTO Directive
• Parameter Classifications
• Example: Exchanging 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:

```assembly
.data
myWord WORD ?
.code
INVOKED mySub, ADDR myWord
```
PROC Directive (1 of 2)

• The PROC directive declares a procedure with an optional list of named parameters.
• Syntax:
  \[ label \text{ PROC} \text{ paramList}\]
• \text{paramList} is a list of parameters separated by commas. Each parameter has the following syntax:
  \[ \text{paramName} : \text{type}\]

\text{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:

\[ label \text{ PROC,} \]
\[ \text{paramList} \]

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

\[ \text{param-1:type-1, param-2:type-2, . . ., param-n:type-n} \]

• Or they can be on separate lines:

\[ \text{param-1:type-1,} \]
\[ \text{param-2:type-2,} \]
\[ \ldots, \]
\[ \text{param-n:type-n} \]
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
```
Trouble-Shooting Tips

• Save and restore registers when they are modified by a procedure.
  – Except a register that returns a function result

• When using INVOKE, be careful to pass a pointer to the correct data type.
  • For example, MASM cannot distinguish between a DWORD argument and a PTR BYTE argument.

• Do not pass an immediate value to a procedure that expects a reference parameter.
  • Dereferencing its address will likely cause a general-protection fault.
What's Next

• Stack Frames
• Recursion
• .MODEL Directive
• INVOKE, ADDR, PROC, and PROTO
• Creating Multimodule Programs
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 (think object-oriented here...)
  • **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.
Sample Program output

Enter a signed integer: -25
Enter a signed integer: 36
Enter a signed integer: 42
The sum of the integers is: +53
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
```
Inspect Individual Modules

- Main
- PromptForIntegers
- ArraySum
- DisplaySum

Custom batch file for assembling and linking.
Review Questions

1. (True/False): Linking OBJ modules is much faster than assembling ASM source files.

2. (True/False): Separating a large program into short modules makes a program more difficult to maintain.

3. (True/False): In a multimodule program, an END statement with a label occurs only once, in the startup module.

4. (True/False): PROTO directives use up memory, so you must be careful not to include a PROTO directive for a procedure unless the procedure is actually called.
Summary

• Stack parameters
  – more convenient than register parameters
  – passed by value or reference
  – ENTER and LEAVE instructions

• Local variables
  – created on the stack below stack pointer
  – LOCAL directive

• Recursive procedure calls itself

• Calling conventions (C, stdcall)

• MASM procedure-related directives
  – INVOKE, PROC, PROTO