8

Advanced Procedures

OUTLINES

- Introduction
- Stack Frames
- Recursion
- INVOKE, ADDR, PROC, and PROTO

INTRODUCTION

- In MASM, subroutines are called ***procedures***.
- Values passed to a subroutine by a calling program are called ***arguments***.
- When the values are received by the called subroutine, they are called ***parameters***.
- Parameters can be passed both in registers and on the stack.

8.2 STACK FRAMES

- A *stack frame* (or *activation record*) is the area of the stack set aside for passed arguments, subroutine return address, local variables, and saved registers.

- The Stack Frame is created by the following steps:

1. Passed arguments are pushed on the stack.
2. The subroutine is called, causing the subroutine return address to be pushed on the stack.
3. As the subroutine begins to execute, EBP is pushed on the stack..

4. EBP is set equal to ESP.
   - From this point on, EBP acts as a base reference for all of the subroutine parameters

5. If there are local variables, ESP is decremented to reserve space for the variables on the stack.

6. If any registers need to be saved, they are pushed on the stack.

- The structure of a stack frame is directly affected by a program's memory model and its choice of argument passing convention.

DISADVANTAGES OF REGISTER PARAMETERS

- In earlier chapters, our subroutines received register parameters.

- Because registers' use is multipurpose, the registers used as parameters must
  - first be pushed on the stack before procedure calls,
  - assigned the values of procedure arguments,
  - and later restored to their original values after the procedure returns.
- Extra Pushes/Pops
- Programmers have to be very careful that each register's PUSH is matched by its appropriate POP.

```
push ebx                               ; save register values
push ecx
push esi

mov esi, OFFSET array                  ; use registers as parameters
mov ecx, LENGTHOF array
mov ebx, TYPE array
call DumpMem

pop esi                                ; restore register values
pop ecx
pop ebx
```

- Stack parameters offer a flexible approach that does not require register parameters. Just before the subroutine call, the arguments are pushed on the stack.

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

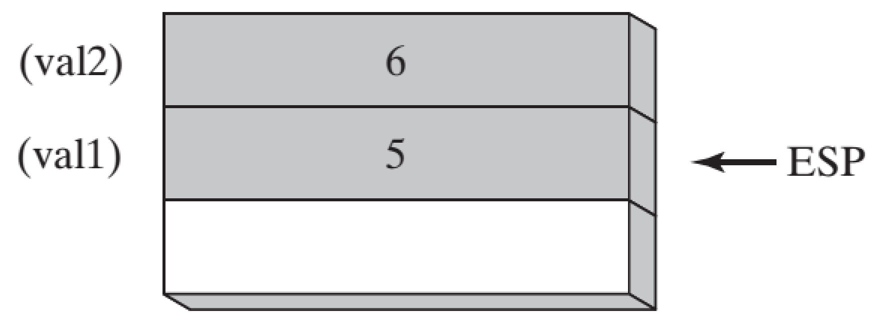
- Two general types of arguments are pushed on the stack during subroutine calls:

1. Value arguments (values of variables and constants)
2. Reference arguments (addresses of variables)

PASSING BY VALUE

- When an argument is passed *by value*, a copy of the value is pushed on the stack.

```
.data
        val1 DWORD 5
        val2 DWORD 6
.code
        push val2
        push val1
        call AddTwo
```



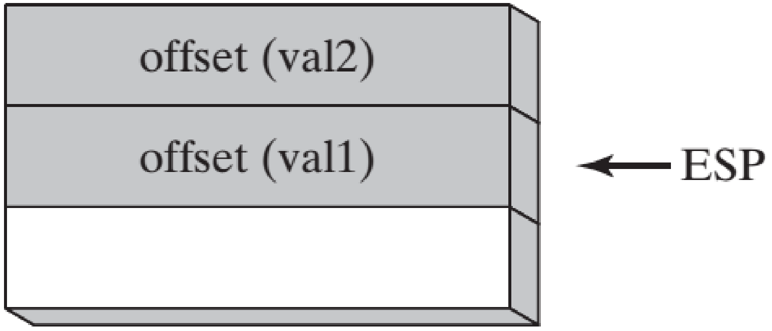
E.g. in C++:

```
int sum = AddTwo( val1, val2 );
```

PASSING BY REFERENCE

- An argument passed by reference consists of the address (offset) of an object

```
push OFFSET val2
push OFFSET val1
call Swap
```



In C++:

```
Swap( &val1, &val2 );
```

PASSING ARRAY

- High-level languages always pass arrays to subroutines by reference. That is, they push the address of an array on the stack. The subroutine can then get the address from the stack and use it to access the array.

```
.data
      array DWORD 50 DUP(?)
.code
      push OFFSET array
      call ArrayFill
```

ACCESSING STACK PARAMETERS

- EBP is called the **base pointer** or **frame pointer** because it holds the base address of the stack frame.
- Given the following C++ function:

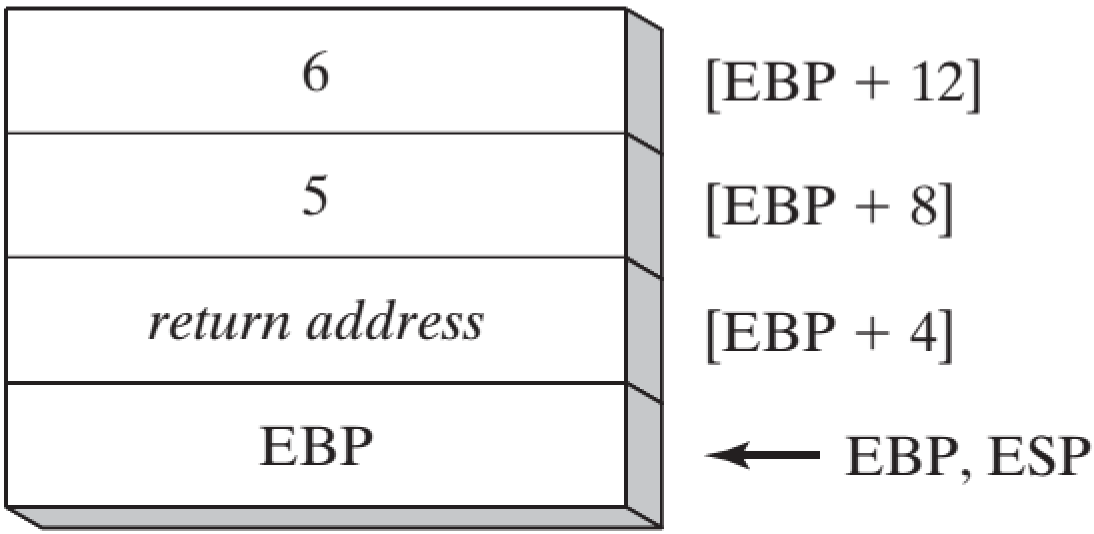
```
int AddTwo( int x, int y )
{
        return x + y;
}
```

- A function call such as **`AddTwo(5,6)`** would cause the second parameter to be pushed on the stack, followed by the first parameter.

- **The Assembly Equivalent:** In its earliest execution, **AddTwo** pushes EBP on the stack to preserve its existing value, and then EBP is set to the same value as ESP, so EBP can be the base pointer for AddTwo's stack frame:

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

- ESP would change value, but EBP would not.
- Base-Offset Addressing is used to access stack parameters.
  - EBP is the base register and the offset is a constant (***explicit stack parameters)*** .



```
AddTwo PROC
       push ebp
       mov ebp,esp          ; base of stack frame
       mov eax,[ebp + 12]   ; second parameter
       add eax,[ebp + 8]    ; first parameter
       pop ebp
       ret
AddTwo ENDP
```

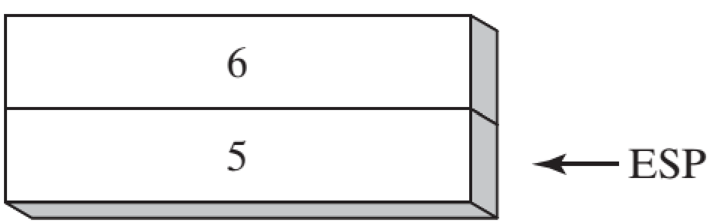
- When stack parameters are referenced with expressions such as `[ebp + 8]`, we call them ***explicit stack parameters.***

CLEANING UP THE STACK

- There must be a way for parameters to be removed from the stack when a subroutine returns.

```
push 6
push 5
call AddTwo
```

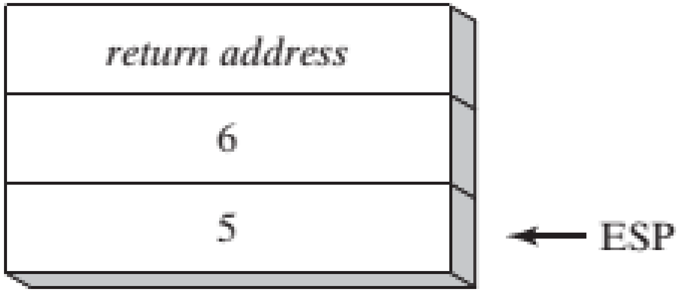
Assuming that `AddTwo` leaves the two parameters on the stack, the following illustration shows the stack after returning from the call:



```
main PROC
 call Example1
 exit
main ENDP

Example1 PROC
  push 6
  push 5
  call AddTwo
  ret                          ; stack is corrupted!
Example1 ENDP
```

- When the RET instruction in Example1 is about to execute, ESP points to the integer 5 rather than the return address that would take it back to main:



- if we were to call AddTwo from a loop, the stack could overflow. Each call uses 12 bytes of stack space:
  - 4 bytes for each parameter, plus 4 bytes for the CALL instruction's return address.

- **32-Bit Calling Conventions**

1. **The C Calling Convention:** The C calling convention is used by the C and C++ programming languages.
   - Subroutine parameters are pushed on the stack in reverse order.

- The C calling convention solves the problem of cleaning up the runtime stack in a simple way:
- When a program calls a subroutine, it follows the CALL instruction with a statement that adds a value to the stack pointer (ESP) equal to the combined sizes of the subroutine parameters.

```
Example1 PROC
        push 6
        push 5
        call AddTwo
        add esp,8       ; remove arguments from the stack
        ret
Example1 ENDP
```

2. **STDCALL Calling Convention:** Another common way to remove parameters from the stack is to use a convention named STDCALL.

- We supply an integer parameter to the RET instruction, which in turn adds integer to ESP after returning to the calling procedure.
- The integer must equal the number of bytes of stack space consumed by the procedure's parameters

```
AddTwo PROC
        push ebp
        mov ebp,esp                     ; base of stack frame
        mov eax,[ebp + 12]              ; second parameter
        add eax,[ebp + 8]               ; first parameter
        pop ebp
        ret 8                           ; clean up the stack
AddTwo ENDP
```

SAVING AND RESTORING REGISTERS

- Subroutines often save the current contents of registers on the stack before modifying them so that the original values can be restored just before the subroutine returns.

```
MySub PROC
       push ebp                    ; save base pointer
       mov ebp,esp                 ; base of stack frame
       push ecx
       push edx                    ; save EDX
       mov eax,[ebp+8]             ; get the stack parameter

       pop edx                     ; restore saved registers
       pop ecx
       pop ebp                     ; restore base pointer
       ret                         ; clean up the stack
MySub ENDP
```

LOCAL VARIABLES

- Local variables are created on the runtime stack, usually below the base pointer (EBP).

```
void MySub()
{
        int X = 10;
        int Y = 20;
}
```

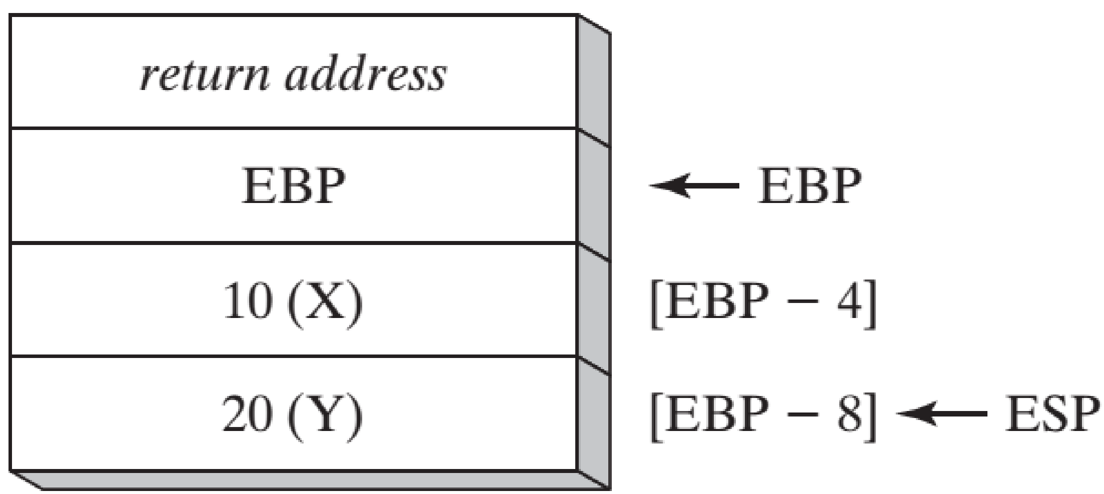
Variable	Bytes	Stack Offset
X	4	EBP – 4
Y	4	EBP – 8

```
MySub PROC

 push ebp
 mov ebp,esp
 sub esp,8                              ; create locals

 mov DWORD PTR [ebp - 4],10             ; X
 mov DWORD PTR [ebp-8],20               ; Y
 mov esp,ebp                            ; remove locals from stack
 pop ebp
 ret

MySub ENDP
```



ENTER AND LEAVE INSTRUCTIONS

- The **ENTER** instruction performs three operations:

1. Pushes `EBP` on the stack (`push ebp`)
2. Sets `EBP` to the base of the stack frame (`mov ebp, esp`)
3. Reserves space for local variables (`sub esp,numbytes`)

ENTER *numbytes, nestinglevel*

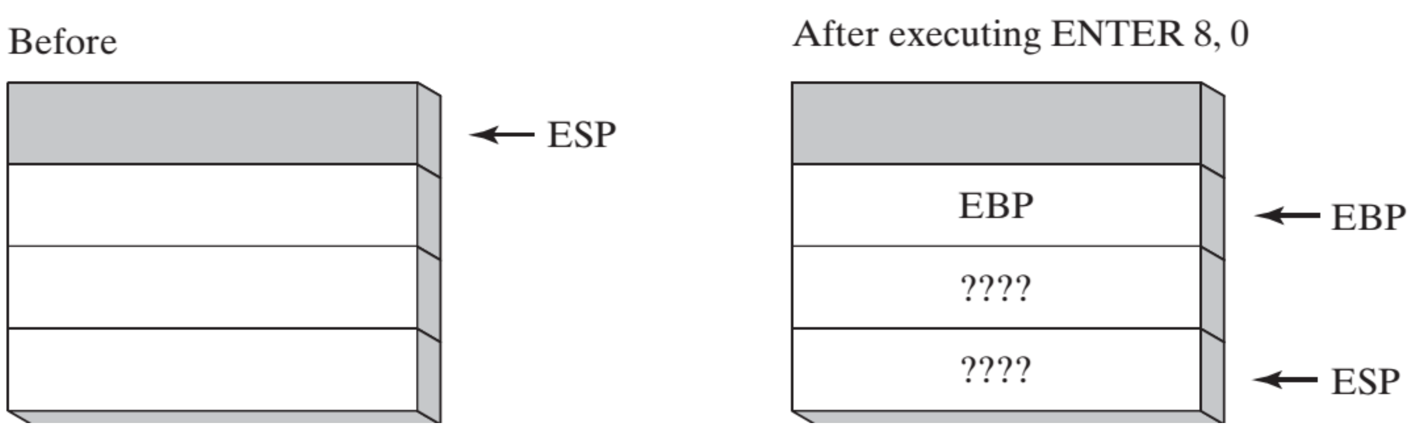
- Both the operands are immediate values,
- The first is a constant specifying the number of bytes of stack space to reserve for local variables
- The second specifies the lexical nesting level of the procedure.

E.g. a procedure with no local variables:

```
MySub PROC
   ENTER 0,0
```

E.g. The `ENTER` instruction reserves 8 bytes of stack space for local variables

```
MySub PROC
   ENTER 8,0
```



- The **LEAVE** instruction terminates the stack frame for a procedure.
  - It reverses the action of a previous ENTER instruction by restoring ESP and EBP to the values they were assigned when the procedure was called.

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

LOCAL DIRECTIVE

LOCAL declares one or more local variables by name, assigning them size attributes.

ENTER, on the other hand, only reserves a single unnamed block of stack space for local variables.

If used, LOCAL must appear on the line immediately following the PROC directive.

```
MySub PROC
     LOCAL var1:BYTE
```

LEA INSTRUCTION

- The LEA instruction returns the address of an indirect operand.
- Ideally suited for use with stack parameters

```
void makeArray()
{
char myString[30];
for( int i = 0; i 30; i++
)
    myString[i] = '*';
}
```

```
makeArray PROC
 push ebp
 mov ebp,esp
 sub esp,32
 lea esi,[ebp-30]
 mov ecx,30
 L1: mov BYTE PTR [esi],'*'
 inc esi
 loop L1
 add esp,32 ; (restore ESP)
 pop ebp
 ret
makeArray ENDP
```

8.3 RECURSION

- A *recursive subroutine* is one that calls itself.

- *Recursion*, the practice of calling recursive subroutines, can be a powerful tool when working with data structures that have repeating patterns.

- Useful recursive subroutines always contain a terminating condition.

- When the terminating condition becomes true, the stack unwinds when the program executes all pending RET instructions.

```
INCLUDE Irvine32.inc

.data
        endlessStr BYTE "This recursion never stops",0

.code
  main PROC
    call Endless
    exit
  main ENDP

  Endless PROC
    mov edx,OFFSET endlessStr
    call WriteString
    call Endless
    ret                              ; never executes
  Endless ENDP

END main
```

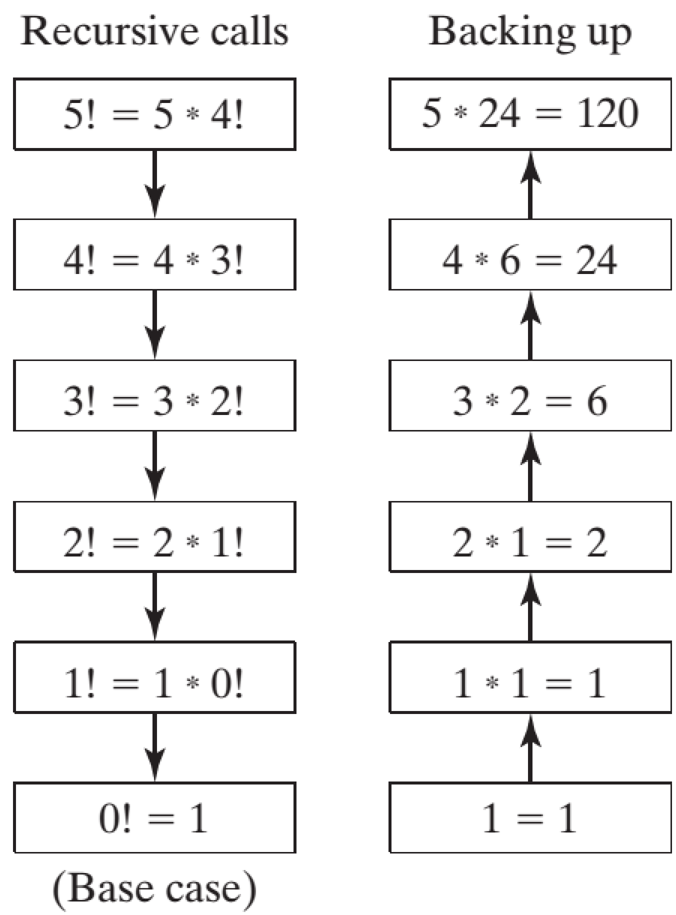
```
INCLUDE Irvine32.inc
.code
main PROC
        mov ecx,5                          ; count = 5
        mov eax,0                          ; holds the sum
        call CalcSum                       ; calculate sum
        L1: call WriteDec                  ; display EAX
        call Crlf                          ; new line
        exit
main ENDP

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
end Main
```

Pushed on Stack	Value in ECX	Value in EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

CALCULATING A FACTORIAL

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



8.4 INVOKE, ADDR, PROC, AND PROTO

- In 32-bit mode, the `INVOKE`, `PROC`, and `PROTO` directives provide powerful tools for defining and calling procedures.

- The `ADDR` operator is an essential tool for defining procedure parameters.

- Their use is controversial because they mask the underlying structure of the runtime stack.

INVOKE DIRECTIVE

- The `INVOKE` directive, only available in 32-bit mode, pushes arguments on the stack and calls a procedure.

- `INVOKE` is a convenient replacement for the `CALL` instruction because it lets you pass multiple arguments using a single line of code.

INVOKE *procedureName [, argumentList]*

CALL VS INVOKE

```
push TYPE array
push LENGTHOF array
push OFFSET array
call DumpArray
```

The equivalent statement using INVOKE is reduced to a single line in which the arguments are listed in reverse order (assuming STDCALL is in effect)

```
INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array
```

INVOKE permits almost any number of arguments, and individual arguments can appear on separate source code lines

Type	Examples
Immediate value	10, 3000h, OFFSET mylist, TYPE array
Integer expression	(10 * 20), COUNT
Variable	myList, array, myWord, myDword
Address expression	[myList+2], [ebx + esi]
Register	eax, bl, edi
ADDR *name*	ADDR myList
OFFSET *name*	OFFSET myList

```
.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

- The ADDR operator, also available in 32-bit mode, can be used to pass a pointer argument when calling a procedure using INVOKE:

```
INVOKE FillArray, ADDR myArray
```

- The ADDR operator can only be used in conjunction with INVOKE:

```
mov esi, ADDR myArray                ; error
```

- The argument passed to ADDR must be an assembly time constant.

```
INVOKE mySub, ADDR [ebp+12]          ; error
```

PROC DIRECTIVE

- The `PROC` directive declares a procedure with an optional list of named parameters.

label **`PROC`**`,` *parameter_list*

- The PROC directive permits you to declare a procedure with a comma-separated list of named parameters.

`label` **`PROC`**`,` *`parameter_1, parameter_2, ..., parameter_n`*

- Your implementation code can refer to the parameters by name rather than by calculated stack offsets such as [ebp - 8].

- A single parameter has the following syntax:

```
paramName: type
```

- *ParamName* is an arbitrary name you assign to the parameter. Its scope is limited to the current procedure (called *local scope*).

- *Type* can be one of : BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, or TBYTE may be a pointer to an existing type (qualified type)

```
PTR BYTE          PTR SBYTE
PTR WORD          PTR SWORD
PTR DWORD         PTR SDWORD
PTR QWORD         PTR TBYTE
```

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

    mov eax,val1
    add eax,val2

    ret
AddTwo ENDP
```

```
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
```

Example 1 The AddTwo procedure receives two doubleword values and returns their sum in EAX:

```
AddTwo PROC,
     val1:DWORD,
     val2:DWORD
     mov   eax,val1
     add   eax,val2
     ret
AddTwo ENDP
```

Example 2 The FillArray procedure receives a pointer to an array of bytes:

```
FillArray PROC,
     pArray:PTR BYTE
     . . .
FillArray ENDP
```

Example 3 The Swap procedure receives two pointers to doublewords:

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

USES OPERATOR

- The USES operator, coupled with the PROC directive, lets you list the names of all registers modified within a procedure.
- USES tells the assembler to do two things:

1. First, generate PUSH instructions that save the registers on the stack at the beginning of the procedure.
2. Second, generate POP instructions that restore the register values at the end of the procedure.

```
ArraySum PROC USES esi ecx
     mov   eax,0
L1:
     add   eax,[esi]
     add   esi,TYPE DWORD
     loop  L1

     ret
ArraySum ENDP
```

```
ArraySum PROC
     push  esi
     push  ecx
     mov   eax,0
L1:
     add   eax,[esi]
     add   esi,TYPE DWORD
     loop  L1
     pop   ecx
     pop   esi
     ret
ArraySum ENDP
```

PROTO DIRECTIVE

- Creates a procedure prototype

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.

- 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
```

Prototype for the ArraySum procedure, showing its parameter list:

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

PARAMETER CLASSIFICATION

- **Input:** 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 limited to the procedure itself.

- **Output:** An output parameter is created when a calling program passes the address of a variable to a procedure.
  - The procedure uses the address to locate and assign data to the variable.

PARAMETER CLASSIFICATION

- **Input–Output**: An input–output parameter is identical to an output parameter, with one exception: The called procedure expects the variable referenced by the parameter to contain some data.
  - The procedure is also expected to modify the variable via the pointer

ARGUMENT SIZE MISMATCH

- Array addresses are based on the sizes of their elements.
  - To address the second element of a doubleword array, for example, one adds 4 to the array's starting address.

- Suppose we call a procedure passing pointers to the first two elements of **DoubleArray.** If we incorrectly calculate the address of the second element as **DoubleArray + 1,** the resulting values after calling that procedure are incorrect:

PASSING THE WRONG TYPE OF POINTER

- When using INVOKE, remember that the assembler does not validate the type of pointer you pass to a procedure.
- Suppose that a procedure (**Swap**) expects to receive two doubleword pointers. and we inadvertently pass it pointers to bytes:

```
.data
        ByteArray BYTE 10h,20h,30h,40h,50h,60h,70h,80h
.code
        INVOKE Swap, ADDR [ByteArray + 0], ADDR [ByteArray + 1]
```

- The program will assemble and run, but 32-bit values are exchanged rather than 8 bits.

PASSING IMMEDIATE VALUES

- If a procedure has a reference parameter, do not pass it an immediate argument.

```
Sub2 PROC, dataPtr:PTR WORD
        mov esi,dataPtr          ; get the address
        mov WORD PTR [esi],0     ; dereference, assign zero
        ret
Sub2 ENDP
```

- The following INVOKE statement assembles but causes a runtime error. The **Sub2** procedure receives 1000h as a pointer value and dereferences memory location 1000h:

```
INVOKE Sub2, 1000h
```

ADVANCED USE OF PARAMETERS

Passing 8-bit and 16-bit Arguments on the Stack

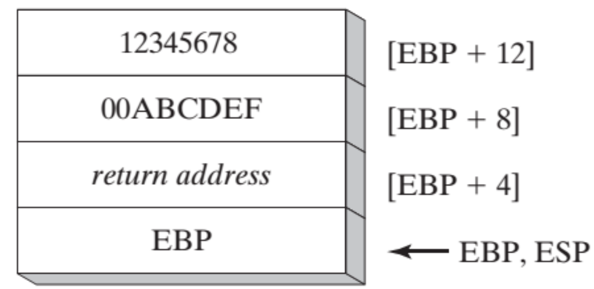
- When passing stack arguments to procedures in 32-bit mode, it’s best to push 32-bit operands.
- Though you can push 16-bit operands on the stack, doing so prevents ESP from being aligned on a doubleword boundary.
- A page fault may occur and runtime performance may be degraded.

PASSING 64-BIT ARGUMENTS

- In 32-bit mode, when passing 64-bit integer arguments to subroutines on the stack, push the high-order doubleword of the argument first, followed by the low-order doubleword.
- Doing so places the integer into the stack in little-endian order (low order byte at the lowest address)

```
.data
longVal QWORD 1234567800ABCDEFh
.code
push  DWORD PTR longVal + 4     ; high doubleword
push  DWORD PTR longVal         ; low doubleword
call  WriteHex64
```

Stack frame after pushing EBP.

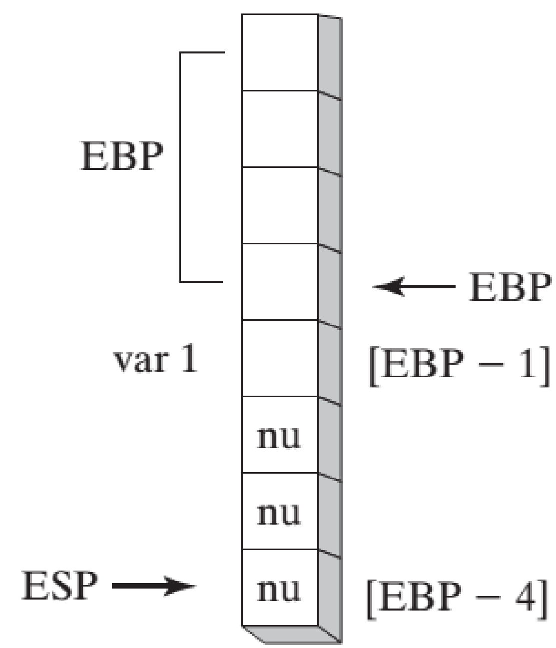



```
WriteHex64 PROC
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+12]          ; high doubleword
    call  WriteHex
    mov   eax,[ebp+8]           ; low doubleword
    call  WriteHex
    pop   ebp
    ret   8
WriteHex64 ENDP
```

NON-DOUBLEWORD LOCAL VARIABLES

- The LOCAL directive has interesting behavior when you declare local variables of differing sizes. Each is allocated space according to its size
  - An 8-bit variable is assigned to the next available byte, a 16-bit variable is assigned to the next even address (word-aligned), and a 32-bit variable is allocated the next doubleword aligned boundary.

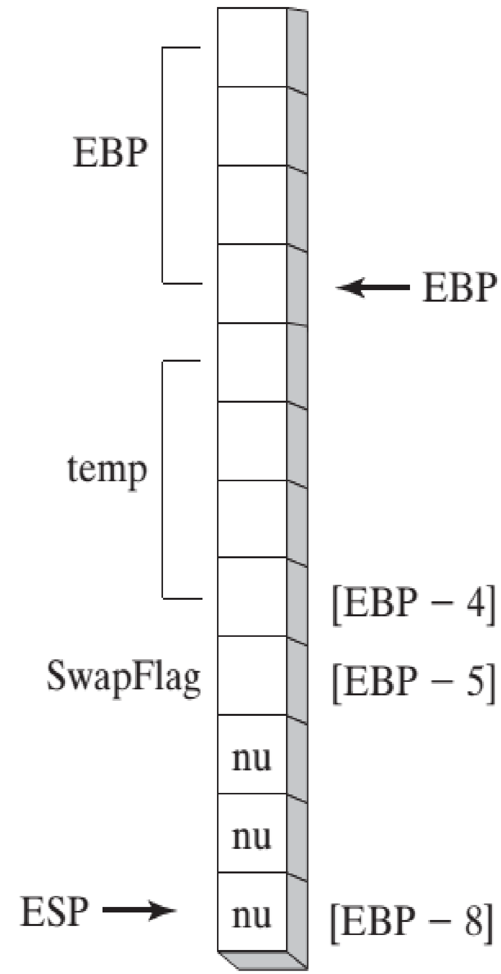
```
Example1 PROC
    LOCAL var1:byte
    mov  al,var1                    ; [EBP - 1]
    ret
Example1 ENDP
```



Because stack offsets default to 32 bits in 32-bit mode, one might expect var1 to be located at EBP – 4.

```
Example2 PROC
    local temp:dword, SwapFlag:BYTE
    .
    .
    ret
Example2 ENDP
```

```
push ebp
mov  ebp,esp
add  esp,0FFFFFFF8h              ; add -8 to ESP
mov  eax,[ebp-4]                 ; temp
mov  bl,[ebp-5]                  ; SwapFlag
leave
ret
```



Although **SwapFlag** is only a byte variable, ESP is rounded downward to the next doubleword stack location

- If you plan to create arrays larger than a few hundred bytes as local variables, be sure to reserve adequate space for the runtime stack, using the STACK directive.

If procedure calls are nested, the runtime stack must be large enough to hold the sum of all local variables active at any point in the program's execution. In the following code, for example, **Sub1** calls **Sub2**, and **Sub2** calls **Sub3**. Each has a local array variable:

```
Sub1 PROC
local array1[50]:dword          ; 200 bytes
callSub2
.
.
ret
Sub1 ENDP

Sub2 PROC
local array2[80]:word           ; 160 bytes
callSub3
.
.
ret
Sub2 ENDP

Sub3 PROC
local array3[300]:dword         ; 1200 bytes
.
.
ret
Sub3 ENDP
```

When the program enters **Sub3**, the runtime stack holds local variables from **Sub1**, **Sub2**, and **Sub3**. The stack will require 1,560 bytes to hold the local variables, plus the two procedure return addresses (8 bytes), plus any registers that might have been pushed on the stack within the procedures.

SUMMARY

- Stack Frames
- Stack Parameters
- Recursion
- Instructions, Operators, and Directives
  - ADDR, ENTER, LEAVE, INVOKE, PROC, PROTO, RET, USES, LOCAL
- Debugging Tips
- Advanced Use of Parameters