Lab 7: Procedures and the Stack

Contents

7.1 Runtime Stack and Stack Instructions

A stack is a LIFO structure (Last-In, First-Out), because the last value put into the stack is always the first value taken out. The **runtime stack** is a memory array managed directly by the CPU. The ESP is called the **extended stack pointer** register and contains the 32-bit address of the last item that was pushed on the stack. We rarely manipulate the ESP register directly. Instead, the PUSH, POP, CALL, and RET instructions modify the ESP register.

The **.STACK** directive allocates a stack in memory. By default, the assembler allocates 1KB of memory for the stack, which is sufficient for small programs. You can also specify the size of the stack in bytes. For example, the following directive allocates a stack of 4096 bytes. The operating system initializes ESP register to the address of the first byte above the stack.

```
.STACK 4096
```

There are several important uses of the stack in programs:

- A stack makes a convenient temporary save area for registers when they are used for more than one purpose. After they are modified, they can be restored to their original values.
- When the CALL instruction executes, the CPU saves the return address on the stack.
- When calling procedures, we often pass parameter values on the stack.
- Local variables inside a procedure are allocated on the stack and discarded when the procedure ends.

7.1.1 PUSH and POP Instructions

The PUSH instruction has the following syntax: **`push source`**

The **source** operand can be 16-bit or 32-bit register, a word or double word in memory, or an immediate constant. For a word-size source operand, the ESP register is decremented by 2 and the source operand is stored on the stack at the address pointed by the ESP register. For a double word-size source operand, the ESP register is decremented by 4 and the source operand is stored. The stack grows downwards or backwards towards lower addresses.

The POP instruction has the following syntax: **`pop destination`**

The **destination** operand can be a word or double word in memory, or a 16-bit or 32-bit general purpose register. The POP instruction does the opposite job a PUSH. It copies the word or double word on top of the stack into the destination memory or register and then increments the ESP register. If the destination is a word then the ESP register is incremented by 2. Otherwise, it is incremented by 4 bytes.

7.1.2 PUSHAD, POPAD, PUSHA, and POPA Instructions

The PUSHAD instruction pushes all of the 32-bit general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI. The POPAD instruction pops the same registers off the stack in reverse order. If you write a procedure that modifies a number of 32-bit registers, then you can use the PUSHAD at the beginning of the procedure to save all the 32-bit general-purpose registers before they are modified, and you use the POPAD at the end of the procedure to restore their values.

Similarly, the PUSHA instruction pushes the 16-bit general-purpose registers in the following order: AX, CX, DX, BX, SP (original value), BP, SI, and DI. The POPA instruction pops the same registers in reverse order.

7.1.3 PUSHFD and POPFD Instructions

There are times when it is useful to save the flags so you can restore their values later. The PUSHFD instruction pushes the 32-bit EFLAGS register on the stack, and POPFD pops the stack into EFLAGS.

7.1.4 Lab Work: Demonstrating the Stack Instructions

```
TITLE Demonstrating Stack Instructions  (stack.asm)

.686
.MODEL flat, stdcall
.STACK 4096
INCLUDE Irvine32.inc

.data
var1    DWORD   01234567h
var2    DWORD   89ABCDEFh

.code
main PROC
    pushad  ; Save general-purpose registers

    ; PUSH and POP
    push var1
    push var2
    push 6A6A4C4Ch
    pop  eax
    pop  ebx
    pop  cx
    pop  dx

    popad   ; restore general-purpose registers

    ; Exchanging 2 variables in memory
    push var1
    push var2
    pop  var1
    pop  var2

    exit
main ENDP
END main
```

Analyze the above program and guess the values of the **eax**, **ebx**, **cx**, and **dx** registers after executing the **pop dx** instruction. Write these values in hexadecimal in the shown boxes:

EAX (hex) =	**EBX (hex) =**
CX (hex) =	**DX (hex) =**

Also guess and write the values of **var1** and **var2** after executing the **pop var2** instruction:

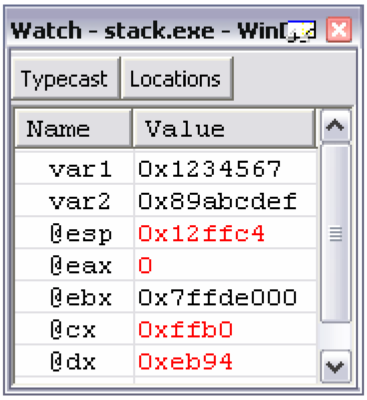
var1 (hex) =	**var2 (hex) =**

7.1.5 Lab Work: Assemble and Link Program *stack.asm*

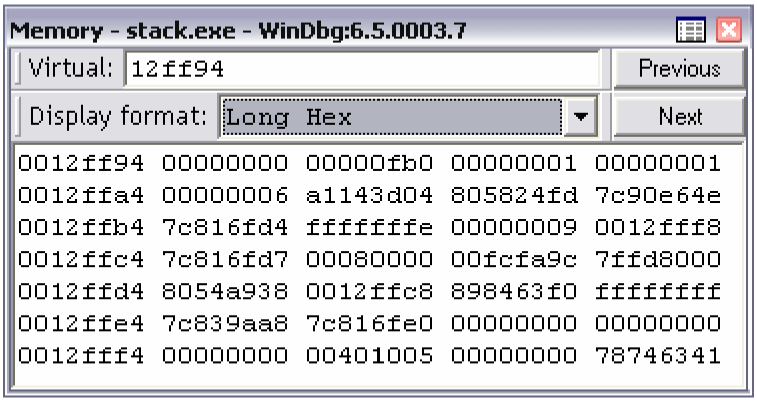
7.1.6 Lab Work: Trace the Execution of Program *stack.exe*

Run the 32-bit Windows Debugger. Open the source file *stack.asm*. Open a Watch window and watch the variables **var1** and **var2**, as well as the registers **esp**, **eax**, **ebx**, **cx**, and **dx**. You should type the **@** symbol before the register name to watch a register in the Watch window. Place the cursor at the beginning of the *main* procedure and press **F7** to start debugging it. Observe the **esp** register is initialized to **0x12ffc4**, or to some other value depending on the program or the computer system. The operating system initialized the **esp** register at the beginning of the program execution.

Open a Memory window to view the stack and type **12ff94** as the virtual address as shown below. A smaller value **12ff94** is chosen than **12ffc4** because the stack grows downwards. This will allow you to view **48** bytes of the stack (0xc4 – 0x94 = 48 bytes). Notice that the stack is uninitialized and contains “junk” values. You can change the Display format to ‘Long Hex’ to view the stack as double words rather than bytes, and resize the window to view 4 double words (or 16 bytes) per line.



Name	Value
var1	0x1234567
var2	0x89abcdef
@esp	0x12ffc4
@eax	0
@ebx	0x7ffde000
@cx	0xffb0
@dx	0xeb94



Virtual: 12ff94

Display format: Long Hex

```
0012ff94 00000000 00000fb0 00000001 00000001
0012ffa4 00000006 a1143d04 805824fd 7c90e64e
0012ffb4 7c816fd4 fffffffe 00000009 0012fff8
0012ffc4 7c816fd7 00080000 00fcfa9c 7ffd8000
0012ffd4 8054a938 0012ffc8 898463f0 ffffffff
0012ffe4 7c839aa8 7c816fe0 00000000 00000000
0012fff4 00000000 00401005 00000000 78746341
```

Now press **F10** to step through the program execution. The first instruction **pushad** will push all the eight 32-bit general-purpose registers on the stack. Observe that the **esp** register has changed to **0x12ffa4,** and that the second and third rows starting at addresses **0012ffa4** and **0012ffb4** have been modified to contain a copy of the register values. Continue pressing **F10** and try to anticipate the changes at each step. Check the values of the registers and variables after executing the pop instructions, and make the necessary corrections.

7.2 Defining and Using Procedures

A procedure is declared using the **PROC** and **ENDP** directives. It must be assigned a name and must end with a return statement. Each program we have written so far contains a procedure called main. As programs become larger, it is important to break them down into procedures to make them more modular and to simplify programming.

As an example, let us write a procedure name *SumOf* that calculates the sum of three 32-bit integers. We will assume that the integers are assigned to EAX, EBX, and ECX before the procedure is called. The procedure returns the sum in EAX:

```
SumOf PROC
      add eax, ebx
      add eax, ecx
      ret
SumOf ENDP
```

7.2.1 CALL and RET Instructions

The **CALL** instruction calls a procedure by directing the processor to begin execution at the first instruction inside the procedure. The procedure uses a **RET** instruction to return from the procedure. This will allow the processor to continue at the instruction that appears just after the **CALL** instruction where the procedure was called.

The **CALL** instruction works as follows:

- It pushes a return address on the stack. This is the address of the instruction appearing after the **CALL** instruction. The return address tells the **RET** instruction where to return.
- It modifies the **EIP** register to contain the address of the called procedure. This is the address of the first instruction inside the called procedure. The processor starts executing the body of the procedure.

The RET instruction works as follows:

- It pops the return address off the stack into the instruction pointer (**EIP** register). The processor will continue program execution at the instruction that appears after the **CALL** instruction.

Before calling a procedure, you have to pass to it the parameters it expects. For example, the above procedure *SumOf* expects three parameters in the **eax**, **ebx**, and **ecx** registers:

```
mov eax, 40000h
mov ebx, 60000h
mov ecx, 80000h
call SumOf
. . . ; result is in the EAX register
```

7.2.2 Demonstrating Procedures

```
TITLE Demonstrating Procedures  (procedure.asm)

.686
.MODEL flat, stdcall
.STACK 4096
INCLUDE Irvine32.inc

.data

.code
main PROC
    mov  eax, 9876
    mov  ebx, 12
    mov  ecx, -5
    call sort3
    exit
main ENDP
```

```
; Sorts 3 integers in EAX, EBX, and ECX
sort3 PROC
    cmp  eax, ebx
    jle  L1
    call swapAB
L1: cmp  eax, ecx
    jle  L2
    call swapAC
L2: cmp  ebx, ecx
    jle  L3
    call swapBC
L3: ret
sort3 ENDP

; Swaps the values of the EAX and EBX registers
swapAB PROC
    Push eax
    Push ebx
    Pop  eax
    Pop  ebx
    ret
swapAB ENDP

; Swaps the values of the EAX and ECX registers
swapAC PROC
    Push eax
    Push ecx
    Pop  eax
    Pop  ecx
    ret
swapAC ENDP

; Swaps the values of the EBX and ECX registers
swapBC PROC
    Push ebx
    Push ecx
    Pop  ebx
    Pop  ecx
    ret
swapBC ENDP
END main
```

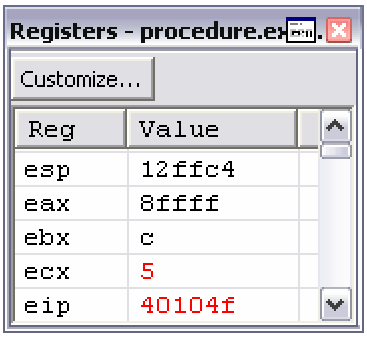
The above program demonstrates the definition and use of procedures in a program. It has five procedures: *main*, *sort3*, *swapAB*, *swapAC*, and *swapBC*. Program execution starts in the *main* procedure. The *main* procedure calls the *sort3* procedure, which calls the *swapAB*, *swapAC*, and *swapBC* procedures, to sort the *EAX*, *EBX*, and *ECX* registers in ascending order. You will now trace the execution of the above program to understand procedure call and return, and how the return address is saved on the stack.

7.2.3 Lab Work: Assemble and Link *procedure.asm*

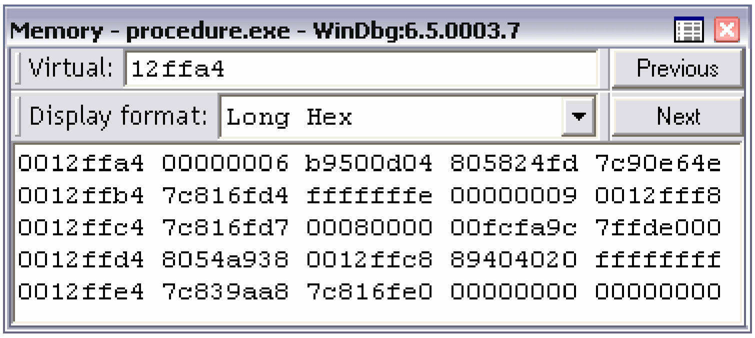
7.2.4 Lab Work: Trace the Execution of *procedure.exe*

Run the 32-bit Windows Debugger. Open the source file *procedure.asm*. Open a Register window and view the registers **esp**, **eax**, **ebx**, **ecx**, and **eip**. The **eip** register is very important here because it contains the address the next instruction to be executed. Place the cursor at the beginning of the *main* procedure and press **F7** to start debugging it. Observe that the **esp** register is initialized to **12ffc4** (hexadecimal) by the operating system.

Open a Memory window to view the stack and type **12ffa4** as the virtual address as shown below. Change the Display format to ‘Long Hex’ to view the stack as double words rather than bytes, and resize the window to view 4 double words (or 16 bytes) per line.



Reg	Value
esp	12ffc4
eax	8ffff
ebx	c
ecx	5
eip	40104f



Virtual: 12ffa4

Display format: Long Hex

```
0012ffa4 00000006 b9500d04 805824fd 7c90e64e
0012ffb4 7c816fd4 fffffffe 00000009 0012fff8
0012ffc4 7c816fd7 00080000 00fcfa9c 7ffde000
0012ffd4 8054a938 0012ffc8 89404020 ffffffff
0012ffe4 7c839aa8 7c816fe0 00000000 00000000
```

At the **call sort3** instruction, press **F8** to step into the procedure call. This will take you to the beginning of the **sort3** procedure. If you press **F10** at **call sort3**, the debugger will execute and return from procedure **sort3**, without showing you the details of the procedure call and return. Therefore, you should use the **F8** key to trace procedure calls and returns.

The **call** instruction will push the return address on the stack, and the **ret** instruction will pop it off the stack. Now answer the following as you trace the procedure calls and returns:

What is the return address of **call sort3** (in hex)? ....

Where is it located on the stack (stack address)? ....

What is the return address of **call swapAB** (in hex)? ....

Where is it located on the stack (stack address)? ....

What is the return address of **call swapAC** (in hex)? ....

Where is it located on the stack (stack address)? ....

What is the return address of **call swapBC** (in hex)? ....

Where is it located on the stack (stack address)? ....

7.3 Stack Parameters

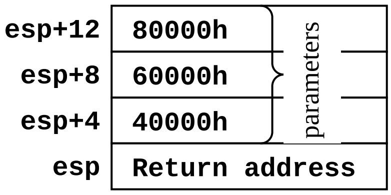
There are two basic ways of passing parameters to procedures. We can pass parameters in registers or we can pass them on the stack. So far, we have demonstrated the use of register parameters in Section 7.2. The Irvine library also uses register parameters for its procedures. Register parameters are optimized for program execution speed.

Stack parameters are pushed on the stack before making the procedure call. The called procedure will have to locate its parameters on the stack. Stack parameters standardizes procedure calls. Nearly all high-level programming languages and libraries use them. If you want to call procedures in the MS-Windows library, you have to use stack parameters.

Consider the *SumOf* procedure defined in Section 7.2. We now redefine it and call it differently, passing parameters on the stack, rather than in registers as shown below:

```
push 80000h
push 60000h
push 40000h
call SumOf
```

```
SumOf PROC
   mov eax, [esp+4]
   add eax, [esp+8]
   add eax, [esp+12]
   ret 12
SumOf ENDP
```



esp+12	80000h
esp+8	60000h
esp+4	40000h
esp	Return address

When the *SumOf* procedure is called, the **esp** register is pointing at the return address on top of the stack. The parameters can be located on the stack at addresses **[esp+4]**, **[esp+8]**, and **[esp+12]**. Recall that the stack grows downwards towards lower addresses.

7.3.1 Cleaning up the Stack Parameters with the RET Instruction

When parameters are pushed on the stack, it is important to clean up the stack upon returning from a procedure. The **ret** instruction can specify an extra integer constant to clean up the parameters on the stack. In the above *SumOf* procedure, **12** bytes are pushed on the stack for three parameters. The **ret 12** instruction is used to increment the **esp** register by **12** bytes, in addition to the **4** bytes for popping the return address.

7.3.2 Saving and Restoring Registers

A procedure might need to use some general-purpose registers to carry its computation. To preserve the values of these registers across a procedure call, it is important to push their values at the beginning of the procedure and pop them at the end, so that the caller program can be sure that none of its own register values are overwritten.

7.3.3 The EBP Register

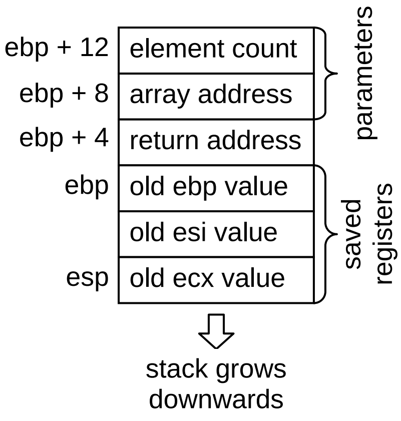
The **esp** register can be used in simple procedures, like the above *SumOf*, to access parameters on the stack. However, in complex procedures, the **esp** register might change. The **ebp** register can be used instead as a base register for accessing parameters on the stack.

7.3.4 Example: Summing an Integer Array

The following procedure computes the sum of an array of integers. The first two instructions save the old value of **ebp** and assign the value of **esp** to **ebp**. These two instructions are commonly used in procedures that have their parameters on the stack. The **ebp** register is used to access the parameters. The first parameter is located at address **[ebp + 8]** and the second parameter is at address **[ebp + 12]**. This is because 8 bytes are allocated on the stack for the return address and the **ebp** register.

The *ArraySum* procedure uses the **esi** register to address the integer array and the **ecx** register to count the number of elements. Since these two registers are modified by *ArraySum*, their values have to be preserved. This is why, the **esi** and **ecx** registers are pushed at the beginning and popped at the end, but in reverse order.

```
ArraySum PROC
   push  ebp           ; save old value of EBP
   mov   ebp, esp      ; EBP locates parameters
   push  esi           ; save old value of ESI
   push  ecx           ; save old value of ECX
   mov   esi, [ebp+8]  ; ESI = array address
   mov   ecx, [ebp+12] ; ECX = number of elements
   mov   eax, 0        ; initialize the sum to 0
   jecxz L2
L1:
   add   eax, [esi]    ; add each element to sum
   add   esi, 4        ; point to next element
   loop  L1            ; sum is in eax register
L2:
   pop   ecx           ; restore value of ECX
   pop   esi           ; restore value of ESI
   pop   ebp           ; restore value of EBP
   ret   8             ; Return and clean up the parameters
ArraySum ENDP
```



The *ArraySum* procedure also uses the **eax** register to accumulate the sum and to hold the result of the procedure. Since the result is returned in the **eax** register, its value should not be preserved by the procedure.

7.3.5 Passing Parameters by Value and by Reference

The *ArraySum* procedure receives two parameters on the stack. The first parameter is the array address and the second parameter is the count of the number of elements. Rather than passing the entire array on the stack, the address is passed only. This is called *pass by reference* because the procedure uses the array address to access the array elements. It is also possible to use the array address to modify the array elements. For example, a procedure that reads an array of integers can use the array address to store the read values. On the other hand, the second parameter, which is the element count, is *passed by value* on the stack.

7.3.6 Lab Work: Complete the *ReadIntArray* Procedure

Open the *ArraySum.asm* file and complete the writing of the *ReadIntArray* procedure. Assemble, link, run, and debug the program to make sure it is working correctly.

7.4 Local Variables and Stack Frames

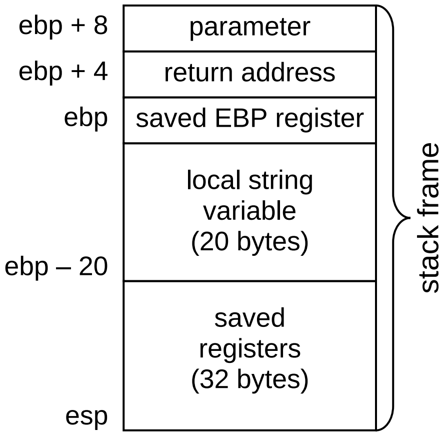
Procedures use local variables to carry the computation. If only few simple local variables are needed, then registers can be used. However, if registers are not enough, then local variables are allocated on the stack when the procedure is called, and freed when the procedure returns.

7.4.1 Example: Taking the Sum of the Digits in a String

```
SumDigits PROC
    push ebp            ; save old EBP value
    mov  ebp, esp       ; new value of EBP
    sub  esp, 20        ; allocate 20 bytes for local string variable
    pushad              ; save general-purpose registers

    lea  edx, prompt    ; write prompt string
    call WriteString
    lea  edx, [ebp-20]  ; EDX = local string address
    mov  ecx, 20        ; maximum chars to read
    call ReadString     ; string is stored on the stack
    mov  ecx, eax       ; save number of chars in ECX

    mov  eax, 0         ; EAX = used to accumulate sum of digits
    mov  ebx, 0         ; BL part of EBX stores one digit
L1: mov  bl, [edx]      ; move one character into BL
    sub  bl, '0'        ; convert character to a number
    cmp  bl, 9          ; check if BL is a digit
    ja   L2             ; skip next instruction if BL is a non-digit
    add  eax, ebx       ; accumulate sum in eax
L2: inc  edx            ; point to next character
    loop L1

    mov  ebx, [ebp+8]   ; EBX = parameter = address of sum
    mov  [ebx], eax     ; store sum indirectly
    popad               ; pop general-purpose registers
    mov  esp, ebp       ; free local string variable
    pop  ebp            ; restore EBP register
    ret  4              ; return and clean the parameter's 4 bytes

    prompt  BYTE    "Enter a string of (max 19) digits: ",0
SumDigits ENDP
```

The *SumDigits* procedure inputs a string of digits and then computes and returns the sum of the string digits. The input string is a local variable stored on the stack.

7.4.2 Stack Frame



The *SumDigits* procedure receives one parameter on the stack. This parameter is the address of the sum variable. After saving the old **ebp** value on the stack and setting the value of **ebp** register, the 3rd instruction **sub esp, 20** allocates **20 bytes** for the local string variable on the stack. To allocate space for local variables on the stack, we simply decrease the value of the **esp** register by the size of the variables in bytes. One **sub** instruction is sufficient to allocate space for all the local variables on the stack.

The **pushad** instruction then saves all the general-purpose registers on the stack to preserve their values. Since many of these registers are used, it is convenient to save all of them at the beginning of the procedure and restore all of them at the end.

The area of the stack, which is allocated for a procedure's parameters, return address, saved registers, and local variables is called a *stack frame*. It is created by the following steps:

- Caller pushes parameters on the stack.
- Procedure is called, the return address is pushed on the stack.
- Procedure saves old EBP value on the stack and sets new EBP equal to ESP.
- A value is subtracted from ESP to create space for the local variables.
- Some or all the general-purpose registers are saved on the stack.

7.4.3 Returning the Result of a Procedure

We have shown you two ways to return the result of a procedure. The first choice is to use one (or more) register to hold the result. For example, we have used the **eax** register to hold the result of the *ArraySum* procedure. The second choice is to use pass by reference to store indirectly the result, as in the *SumDigits* procedure. The **mov [ebx], eax** instruction saves the accumulated sum at the address which was passed as a parameter on the stack.

A final remark about the *SumDigits* procedure is that the *prompt* string is defined at the end of the procedure. The *prompt* string characters are stored after the procedure code, but they will never be executed because they appear after the **ret** instruction.

7.4.4 Lab Work: Assemble, Link, and Trace Program *SumDigits.asm*

Run the 32-bit Windows Debugger to trace the execution of the *SumDigits* procedure. Open a Register window to view the registers and a Memory window to view the stack. Now answer the following just after executing the **pushad** instruction in the *SumDigits* procedure:

What is the value of the parameter on the stack (in hex)? ....

What is the value of the return address on the stack (in hex)? ....

What is the value of the **ebp** register (in hex)? ....

What is the value of the **esp** register (in hex)? ....

How many bytes are allocated for the stack frame? ....

7.5 PROTO, PROC, INVOKE, LOCAL, and USES Directives

MASM defines a number of directives to simplify the writing of procedures.

7.5.1 PROTO and PROC Directives

The PROTO directive declares a prototype for an existing procedure. A prototype specifies a procedure’s name and parameter list. It allows you to call a procedure before defining it. MASM requires a prototype for each procedure called by an INVOKE statement. Consider the *SumDigits* procedure, you can write a prototype for it as follows:

```
SumDigits PROTO sumaddr:PTR DWORD
```

The *SumDigits* prototype specifies one parameter *sumaddr* which is of type PTR DWORD. Once you have defined a prototype for a procedure, MASM requires that you define the procedure to match its prototype as follows:

```
SumDigits PROC sumaddr:PTR DWORD
  ...
  ret
SumDigits ENDP
```

When you specify the parameters of a procedure, the assembler will automatically generate the first two instructions: **push epb** and **mov ebp, esp**. So, there is no need to write them. The assembler will also insert a **leave** instruction just before the **ret** instruction. The **leave** instruction is equivalent to two instructions: **move esp, ebp** and **pop ebp**. So, there is no need to write them either. The assembler will also replace the **ret** instruction with **ret *n***, where ***n*** is equal to the size of the parameters in bytes, so there is no need to specify ***n***. You can also refer to a parameter by name, rather than by its address inside the procedure (e.g., you can replace **[ebp+8]** by **sumaddr**). The assembler will replace the parameter name by its address.

7.5.2 INVOKE Directive

The INVOKE directive simplifies a procedure call by allowing you to pass parameters to a procedure in a single statement. You can invoke *SumDigits* as follows:

```
INVOKE SumDigits, ADDR sum
```

The MASM assembler will generate the following equivalent instructions:

```
push OFFSET sum
call SumDigits
```

If there are multiple parameters, the last parameter is pushed first on the stack and the first parameter is pushed last. This is in accordance to the **stdcall** language specifier.

7.5.3 LOCAL Directive

The LOCAL directive allows you to declare one or more local variables inside a procedure. It must be placed on the line immediately following the PROC directive, as follows:

```
SumDigits PROC sumaddr:PTR DWORD
  LOCAL s[20]:BYTE
  ...
  ret
SumDigits ENDP
```

The assembler will compute the total number of bytes occupied by the local variables and will generate the **add esp, – n** instruction to allocate **n** bytes on the stack. You can refer to local variables declared by the LOCAL directive by name rather than by address inside a procedure. The assembler will substitute the name with its corresponding address.

7.5.4 USES Directive

The USES directive lets you list the names of the all the registers that you want to preserve within a procedure. This directive tells the assembler to generate **push** instructions to save these registers at the beginning, and to generate **pop** instructions that restore these registers at the end of the procedure. You can couple USES with the PROC directive as follows:

```
SumDigits PROC USES eax ebx ecx edx, sumaddr:PTR DWORD
```

7.5.5 Lab Work: Examining the Code Generated by the Assembler

The *SumDigits2.asm* program is a modified version of *SumDigits.asm* that uses assembler directives to simplify the writing of a procedure. Assemble and link *SumDigits2.asm*. Open the *SumDigits2.lst* file and examine the instructions marked with ***** inserted by the assembler at the beginning and end of the *SumDigits* procedure. Write down these instructions:

Instructions inserted at the beginning	Instructions inserted at the end

Review Questions

1. (*True/False*) The **push** instruction decreases the **esp** register and **pop** increases it.
2. How does the **call** instruction work?
3. How does the **ret *n*** instruction work (where ***n*** is an integer constant)?
4. Why is it better to use the **ebp** than the **esp** register to locate parameters on the stack?
5. Which instruction should be used to allocate local variables on the stack?
6. How does the **leave** instruction work?
7. What is the use of an INVOKE directive, and how is it translated?
8. What is the use of a LOCAL directive, and how is it translated?
9. What is the use of a USES directive and how is it translated?

Programming Exercises

1. Write a procedure that fills an array with random integers in the range 0 to 999. The procedure should receive two parameters: the address of the array to be filled in the EAX register, and the count of the elements in the ECX register. Test this procedure separately by calling it from the main procedure.
2. Write a procedure to display an array of integers. The procedure should receive two parameters on the stack: the array address and the count of the elements to be displayed. Test this procedure separately by calling it from the main procedure.
3. Write a procedure to sort an array of integers. The procedure should receive two parameters on the stack: the array address and the count of its elements. To test this procedure, call the array fill procedure (exercise 1) to generate a random array of integers and then call the sort procedure to sort it. Any sorting algorithm may be used.