Chapter 5

Procedures

Chapter 5

Procedures

5.1 Introduction 111

- You can think of several good reasons for you to read this chapter:
  - You want to know how **input-output** works in assembly language.
  - You need to learn about the runtime stack, the fundamental mechanism **for calling and returning from subroutines**.
  - You will learn how to divide large programs into modular subroutines.
  - You will learn about **flowcharts**, which are graphing tools that portray program logic.

5.2 Linking to an External Library 111

5.2.1 Background Information 112

- Link Library Overview
  - A file containing procedures (subroutines) that have been compiled into machine code
    - constructed from one or more OBJ files
- Example:
  - Suppose a program displays a string in the console window by calling a procedure name WriteString.
    - The program source must contain a PROTO directive indentifying the WriteString procedure:

```
WriteString PROTO
```

    - Next, a CALL instruction executes WriteString:

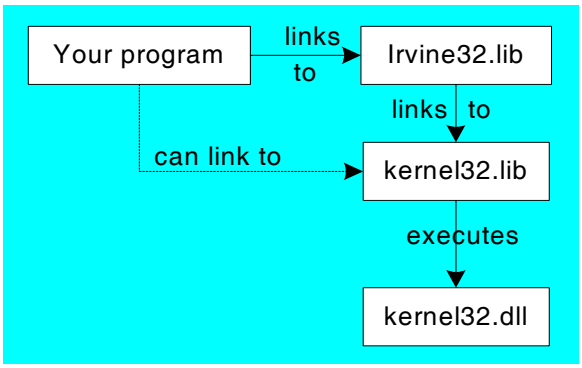
```
call WriteString
```

  - When the program is assembled, the **assembler** leaves the target address of the CALL instruction **blank**, knowing that it will be filled in by the linker.
  - The **linker** looks for WriteString in the link library and **copies** the appropriate machine instructions from the library into the program's executables file. In addition, it inserts WriteString's **address** into the CALL instruction.

- Linking to a Library
  - Linker Command Options
    - The linker utility combines a program's object file with one or more object files and link libraries. The following command, for example, links hello.obj to the irvine32.lib and kernel32.lib libraries:

```
link hello.obj irvine32.lib kernel32.lib
```

  - Linking 32-Bit Programs
    - **kernel32.lib**: Part of the MS-Windows *Platform Software Developments Kit (SDK)*. It contains **linking information** for system functions located in a file named kernel32.dll.
    - **kernel32.dll**: MS-Windows *Dynamic Link Library (DLL)*. It contains **executable functions** that perform character-based input-output.
    - The following figure shows how kernel32.lib is a bridge to kernel32.dll:



5.3 The Book's Link Library 113

5.3.1 Overview 113

- ***Library Procedures - Overview***
  - *CloseFile* Closes an open disk file
  - *Clrscr* Clears console, locates cursor at upper left corner
  - *CreateOutputFile* Creates new disk file for writing in output mode
  - *Crlf* Writes end of line sequence to standard output
  - *Delay* Pauses program execution for *n* millisecond interval
  - *DumpMem* Writes block of memory to standard output in hex
  - *DumpRegs* Displays general-purpose registers and flags (hex)
  - *GetCommandtail* Copies command-line args into array of bytes
  - *GetMaxXY* Gets number of cols, rows in console window buffer
  - *GetMseconds* Returns milliseconds elapsed since midnight
  - *GetTextColor* Returns active foreground and background text colors in the console window
  - *Gotoxy* Locates cursor at row and column on the console
  - *IsDigit* Sets Zero flag if AL contains ASCII code for decimal digit (0–9)
  - *MsgBox* Display popup message boxes
  - *MsgBoxAsk* Display a yes/no question in a popup message box
  - *OpenInputFile* Opens existing file for input
  - *ParseDecimal32* Converts unsigned integer string to binary
  - *ParseInteger32* Converts signed integer string to binary
  - *Random32* Generates 32-bit pseudorandom integer in the range 0 to FFFFFFFFh
  - *Randomize* Seeds the random number generator
  - *RandomRange* Generates a pseudorandom integer within a specified range
  - *ReadChar* Reads a single character from standard input
  - *ReadFromFile* Reads input disk file into buffer
  - *ReadDec* Reads 32-bit unsigned decimal integer from keyboard
  - *ReadHex* Reads 32-bit hexadecimal integer from keyboard
  - *ReadInt-* Reads 32-bit signed decimal integer from keyboard
  - *ReadKey* Reads character from keyboard input buffer
  - *ReadString* Reads string from standard input, terminated by [Enter]
  - *SetTextColor* Sets foreground and background colors of all subsequent console text output
  - *StrLength* Returns length of a string
  - *WaitMsg* Displays message, waits for Enter key to be pressed
  - *WriteBin* Writes unsigned 32-bit integer in ASCII binary format.
  - *WriteBinB* Writes binary integer in byte, word, or doubleword format
  - *WriteChar* Writes a single character to standard output
  - *WriteDec* Writes unsigned 32-bit integer in decimal format
  - *WriteHex* Writes an unsigned 32-bit integer in hexadecimal format
  - *WriteHexB* Writes byte, word, or doubleword in hexadecimal format
  - *WriteInt* Writes signed 32-bit integer in decimal format
  - *WriteString* Writes null-terminated string to console window
  - *WriteToFile* Writes buffer to output file
  - *WriteWindowsMsg* Displays most recent error message generated by MS-Windows

5.3.2 Individual Procedure Descriptions 115

- Example 1
  - Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags

```
.code
call Clrscr
mov  eax,500
call Delay
call DumpRegs
```

Sample output:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```

- Example 2
  - Display a null-terminated string and move the cursor to the beginning of the next screen line

```
.data
str1 BYTE "Assembly language is easy!",0
.code
mov  edx,OFFSET str1
call WriteString
call Crlf
```

- Example 2a
  - Display a null-terminated string and move the cursor to the beginning of the next screen line (use embedded CR/LF)

```
.data
str1 BYTE "Assembly language is easy!",0Dh,0Ah,0
.code
mov  edx,OFFSET str1
call WriteString
```

- Example 3
  - Display an unsigned integer in binary, decimal, and hexadecimal, each on a separate line

```
IntVal = 35
.code
mov  eax,IntVal
call WriteBin         ; display binary
call Crlf
call WriteDec         ; display decimal
call Crlf
call WriteHex         ; display hexadecimal
call Crlf
```

Sample output:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
23
```

- Example 4 - Input a string from the user
  - EDX points to the string
  - ECX specifies the maximum number of characters the user is permitted to enter

```
.data
fileName BYTE 80 DUP(0)
.code
mov edx,OFFSET fileName
mov ecx,SIZEOF fileName – 1
call ReadString
```

Note: A null byte is automatically appended to the string

- Example 5
  - Generate and display ten pseudorandom signed integers in the range 0 – 99
  - Pass each integer to WriteInt in EAX and display it on a separate line

```
.code
mov ecx,10          ; loop counter
L1:  mov  eax,100; ceiling value
   call RandomRange ; generate random int
   call WriteInt    ; display signed int
   call Crlf        ; goto next display line
   loop L1          ; repeat loop
```

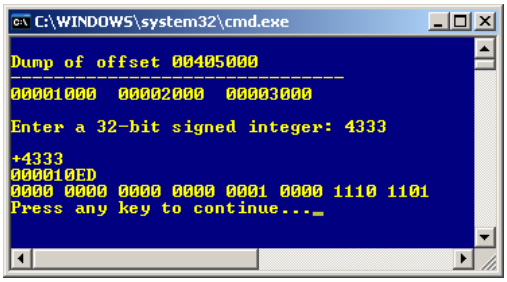
- Example 6
  - Display a null-terminated string with yellow characters on a blue background

```
.data
str1 BYTE "Color output is easy!",0

.code
mov  eax,yellow + (blue * 16)
call SetTextColor
mov  edx,OFFSET str1
call WriteString
call Crlf
```

Note: The background color is multiplied by 16 before being added to the foreground color

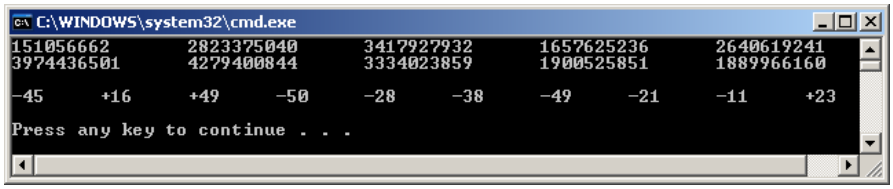
5.3.3 Library Test Programs 125

- Test Program #1: Integer I/O
  - Test program #1 changes the text color to yellow characters on a blue background, dumps an array in hexadecimal, prompts the user for a signed integer, and redisplays the integer in decimal, hexadecimal, and binary:




```
TITLE Library Test #1: Integer I/O   (TestLib1.asm)

; Tests the Clrscr, Crlf, DumpMem, ReadInt,
; SetTextColor, WaitMsg, WriteBin, WriteHex,
; and WriteString procedures.
; Last update: 06/01/2006

INCLUDE Irvine32.inc
.data
arrayD     DWORD 1000h,2000h,3000h
prompt1    BYTE "Enter a 32-bit signed integer: ",0
dwordVal   DWORD ?

.code
main PROC
; Set text color to yellow text on blue background:
   mov  eax,yellow + (blue * 16)
   call SetTextColor
   call Clrscr               ; clear the screen

; Display the array using DumpMem.
   mov  esi,OFFSET arrayD    ; starting OFFSET
   mov  ecx,LENGTHOF arrayD  ; number of units in dwordVal
   mov  ebx,TYPE arrayD      ; size of a doubleword
   call DumpMem              ; display memory
   call Crlf                 ; new line

; Ask the user to input a signed decimal integer.
   mov  edx,OFFSET prompt1
   call WriteString
   call ReadInt              ; input the integer
   mov  dwordVal,eax         ; save in a variable

; Display the integer in decimal, hexadecimal, and binary.
   call Crlf                 ; new line
   call WriteInt             ; display in signed decimal
   call Crlf
   call WriteHex             ; display in hexadecimal
   call Crlf
   call WriteBin             ; display in binary
   call Crlf
   call WaitMsg              ; "Press any key..."

; Return console window to default colors.
   mov  eax,lightGray + (black * 16)
   call SetTextColor
   call Clrscr
   exit
main ENDP
END main
```

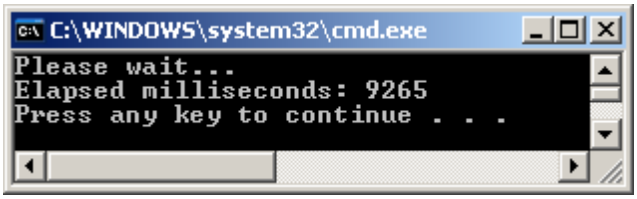
- Test Program #2: Random Integer
  - First, it randomly generates 10 unsigned integers in the range 0 to 4,294,967,294. Next, it generates 10 signed integers in the range -50 to +49.




```
TITLE Link Library Test #2 (TestLib2.asm)

; Testing the Irvine32 Library procedures.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

TAB = 9    ; ASCII code for Tab

.code
main PROC
   call Randomize       ; init random generator
   call Rand1
   call Rand2
   exit
main ENDP

Rand1 PROC
; Generate ten pseudo-random integers.
   mov  ecx,10          ; loop 10 times

L1:  call Random32      ; generate random int
   call WriteDec        ; write in unsigned decimal
   mov  al,TAB          ; horizontal tab
   call WriteChar       ; write the tab
   loop L1

   call Crlf
   ret
Rand1 ENDP

Rand2 PROC
; Generate ten pseudo-random integers between -50 and +49
   mov  ecx,10          ; loop 10 times

L1:  mov  eax,100       ; values 0-99
   call RandomRange     ; generate random int
   sub  eax,50          ; vaues -50 to +49
   call WriteInt        ; write signed decimal
   mov  al,TAB          ; horizontal tab
   call WriteChar       ; write the tab
   loop L1

   call Crlf
   ret
Rand2 ENDP

END main
```

- Test Program #3: Performance Timing
  - The GetMseconds procedure from the link library returns the number of milliseconds elapsed since midnight.
  - In the third test program, we call GetMseconds and execute a nested loop approximately 17 billillion times. After the loop, we call GetMseconds a second time and report the total elapsed time:

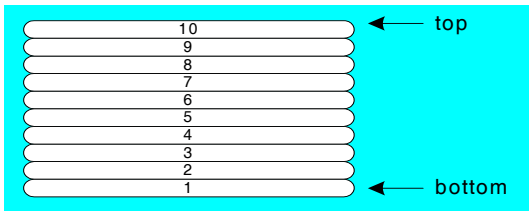
```
TITLE Link Library Test #3    (TestLib3.asm)

; Calculate the elapsed time of executing a nested loop
; about 17 billion times.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

OUTER_LOOP_COUNT = 3    ; adjust for processor speed
.data
startTime DWORD ?
msg1 BYTE "Please wait...",0dh,0ah,0
msg2 BYTE "Elapsed milliseconds: ",0

.code
main PROC
   mov  edx,OFFSET msg1
   call WriteString

; Save the starting time.
   call GetMSeconds
   mov  startTime,eax
   mov  ecx,OUTER_LOOP_COUNT

; Perform a busy loop.
L1:  call innerLoop
   loop L1

; Display the elapsed time.
   call GetMSeconds
   sub  eax,startTime
   mov  edx,OFFSET msg2
   call WriteString
   call WriteDec
   call Crlf
   exit
main ENDP

innerLoop PROC
   push ecx
   mov  ecx,0FFFFFFFFh
L1:  mov  eax,eax
   loop L1
   pop  ecx
   ret
innerLoop ENDP

END main
```



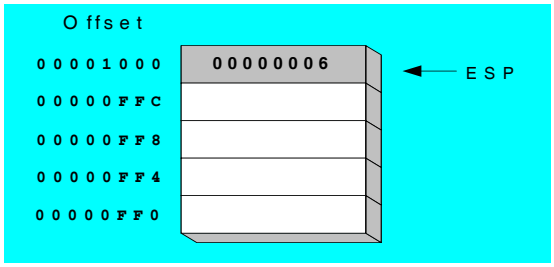
5.4 Stack Operations 129

5.4.1 Runtime Stack 129

- ***Runtime Stack***
  - Imagine a stack of plates . . .
    - plates are only added to the top
    - plates are only removed from the top
    - LIFO (Last-In, First-Out) structure

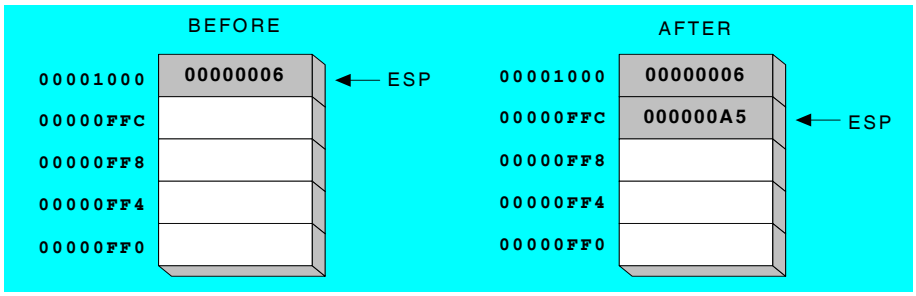


  - Runtime Stack managed by the CPU, using two registers
    - *SS (stack segment)*
    - *ESP (stack pointer)*

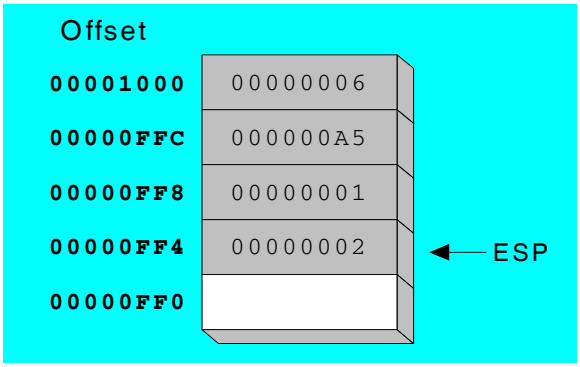


5.4.2 PUSH and POP Instructions 131

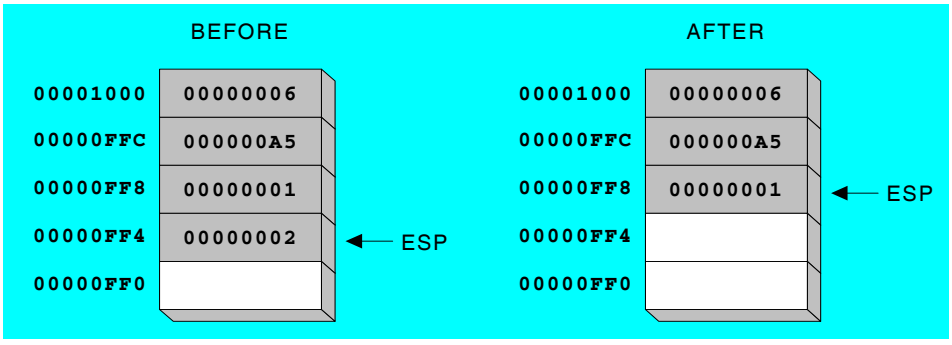
- PUSH Operation (32-bit)
  - Decrement the stack pointer (ESP) by 4
  - Copy a value into the location pointed to by the stack pointer



  - Same stack after pushing two more integers:
    - The stack grows downward



- POP Operation (32-bit)
  - Copy value at stack[ESP] into a register or variable
  - Add 4 to ESP



- PUSH and POP Instructions
  - PUSH syntax:
    - PUSH *r/m16*
    - PUSH *r/m32*
    - PUSH *imm32*
  - POP syntax:
    - POP *r/m16*
    - POP *r/m32*
- Using PUSH and POP
  - Save and restore registers when they contain important values
  - PUSH and POP instructions occur in the opposite order

```
    push esi          ; push registers
    push ecx
    push ebx
    mov  esi,OFFSET dwordVal     ; display some memory
    mov  ecx,LENGTHOF dwordVal
    mov  ebx,TYPE dwordVal
    call DumpMem
    pop  ebx          ; restore registers
    pop  ecx
    pop  esi
```

- Example: Nested Loop
  - When creating a nested loop, push the outer loop counter before entering the inner loop:

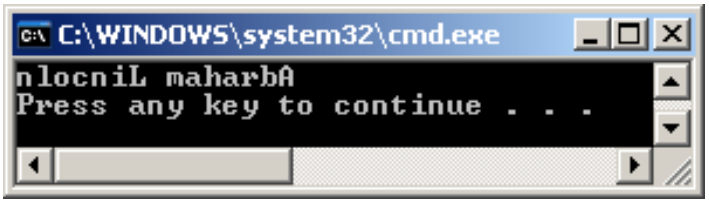
```
      mov ecx,100    ; set outer loop count
    L1:              ; begin the outer loop
      push ecx       ; save outer loop count
      mov ecx,20     ; set inner loop count
    L2:              ; begin the inner loop
        ;
        ;
      loop L2        ; repeat the inner loop
      pop ecx        ; restore outer loop count
      loop L1        ; repeat the outer loop
```

- Related Instructions
  - PUSHFD and POPFD
    - Push and pop the EFLAGS register
  - PUSHAD pushes the 32-bit general-purpose registers on the stack
    - order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
  - POPAD pops the same registers off the stack in reverse order
    - PUSHA and POPA do the same for 16-bit registers

- Example: Reversing a String
  - Use a loop with indexed addressing
  - Push each character on the stack
  - Start at the beginning of the string, pop the stack in reverse order, insert each character back into the string

  Note: only word (16-bit) or doubleword (32-bit) values can be pushed on the stack, each character must be put in EAX before it is pushed

```
TITLE Reversing a String          (RevStr.asm)

; This program reverses a string.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1

.code
main PROC

; Push the name on the stack.
   mov ecx, nameSize
   mov esi, 0

L1:  movzx eax, aName[esi] ; get character
   push eax                ; push on stack
   inc esi
   loop L1

; Pop the name from the stack, in reverse,
; and store in the aName array.
   mov ecx, nameSize
   mov esi, 0

L2:  pop eax               ; get character
   mov aName[esi], al      ; store in string
   inc esi
   loop L2

; Display the name.
   mov  edx,OFFSET aName
   call Writestring
   call Crlf

   exit
main ENDP
END main
```



5.5 Defining and Using Procedures 134

5.5.1 PROC Directive 134

- Creating Procedures
  - Large problems can be divided into smaller tasks to make them more manageable
  - A procedure is the ASM equivalent of a Java or C++ function
  - Following is an assembly language procedure named ***sample***:

```
sample PROC
.
.
ret
sample ENDP
```

- Documenting Procedures
  - A description of all tasks accomplished by the procedure
  - **Receives**: A list of input parameters; state their usage and requirements
  - **Returns**: A description of values returned by the procedure
  - **Requires**: Optional list of requirements called *preconditions* that must be satisfied before the procedure is called
    Note: If a procedure is called without its preconditions satisfied, it will probably not produce the expected output
- Example: SumOf Procedure

```
;-----------------------------------------------------
SumOf PROC
;
; Calculates and returns the sum of three 32-bit integers.
; Receives: EAX, EBX, ECX, the three integers. May be signed or
; unsigned.
; Returns: EAX = sum, and the status flags (Carry, Overflow, etc.)
; are changed.
; Requires: nothing
;-----------------------------------------------------
add eax,ebx
add eax,ecx
ret
SumOf ENDP
```

5.5.2 CALL and RET Instructions 136

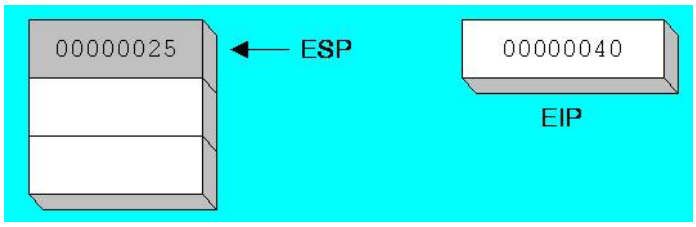
- CALL and RET Instructions
  - The *CALL* instruction calls a procedure
    - pushes offset of **next** instruction on the stack
    - copies the address of the called procedure into **EIP**
  - The *RET* instruction returns from a procedure
    - pops top of stack into EIP

```
main PROC
00000020  call MySub
00000025  mov eax,ebx
          .
          .
main ENDP

MySub PROC
00000040  mov eax,edx
          .
          .
          ret
MySub ENDP
```

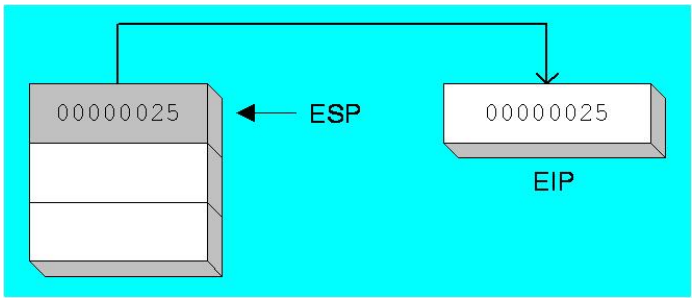
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

- CALL-RET Example



The CALL instruction pushes 00000025 onto the stack, and loads 00000040 into EIP

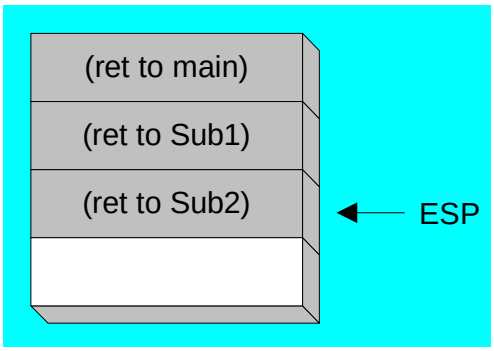


The RET instruction pops 00000025 from the stack into EIP

- Nested Procedure Calls

```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses:



5.5.3 Example: Summing and Integer Array 139

- This version of ArraySum returns the *sum of any doubleword* array whose address is in ***ESI,*** the number of array elements is in ***ECX***, and the sum is returned in ***EAX***:

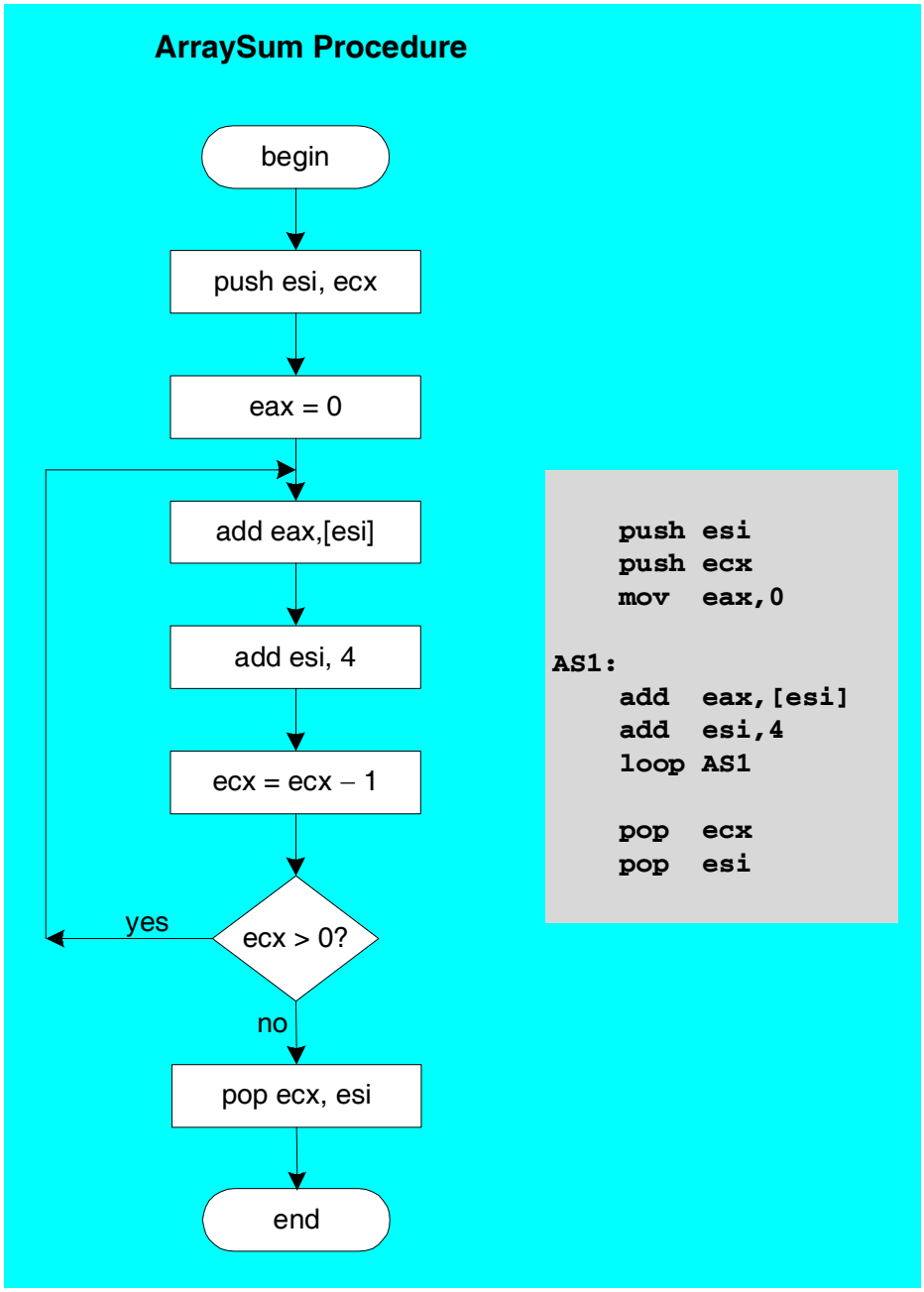
```
ArraySum PROC
; Receives: ESI points to an array of doublewords,
;   ECX = number of array elements.
; Returns: EAX = sum
;-------------------------------------------------
   push   esi            ; save ESI, ECX
   push   ecx
   mov    eax,0          ; set the sum to zero
L1: add    eax,[esi]      ; add each integer to sum
   add    esi,TYPE DWORD ; point to next integer
   loop   L1             ; repeat for array size

   pop    ecx            ; restore ESI, ECX
   pop    esi
   ret                   ; sum is in EAX
ArraySum ENDP
```

- Calling ArraySum

```
.data
array  DWORD  10000h, 20000h, 30000h, 40000h, 50000h
theSum DWORD  ?

.code
main PROC
   mov    esi,OFFSET array   ; ESI points to array
   mov    ecx,LENGTHOF array ; ECX = array count
   call   ArraySum           ; calculate the sum
   mov    theSum, EAX        ; returned in EAX
   exit
main ENDP
```

5.5.4 Flowcharts 140

- ***Flowchart***
  - ***Flowchart*** is a well-established way of *diagramming program logic*
    - Each shape in the flowchart represents a single logic step
    - Lines with arrows connecting the shapes show the ordering of the logical steps
- Flowchart for the ArraySum Procedure



5.5.5 Saving and Restoring Registers 140

- *USES Operator:* Saving and Restoring Registers
  - Lists the registers that will be preserved

```
ArraySum PROC USES ESI ECX
   mov    eax,0          ; set the sum to zero
L1: add    eax,[esi]      ; add each integer to sum
   add    esi,TYPE DWORD ; point to next integer
   loop   L1             ; repeat for array size
   ret                   ; sum is in EAX
ArraySum ENDP
```

MASM generates the code shown in bold:

```
ArraySum PROC
   push   esi            ; save ESI, ECX
   push   ecx
   mov    eax,0          ; set the sum to zero
L1: add    eax,[esi]      ; add each integer to sum
   add    esi,TYPE DWORD ; point to next integer
   loop   L1             ; repeat for array size

   pop    ecx            ; restore ESI, ECX
   pop    esi
   ret                   ; sum is in EAX
ArraySum ENDP
```

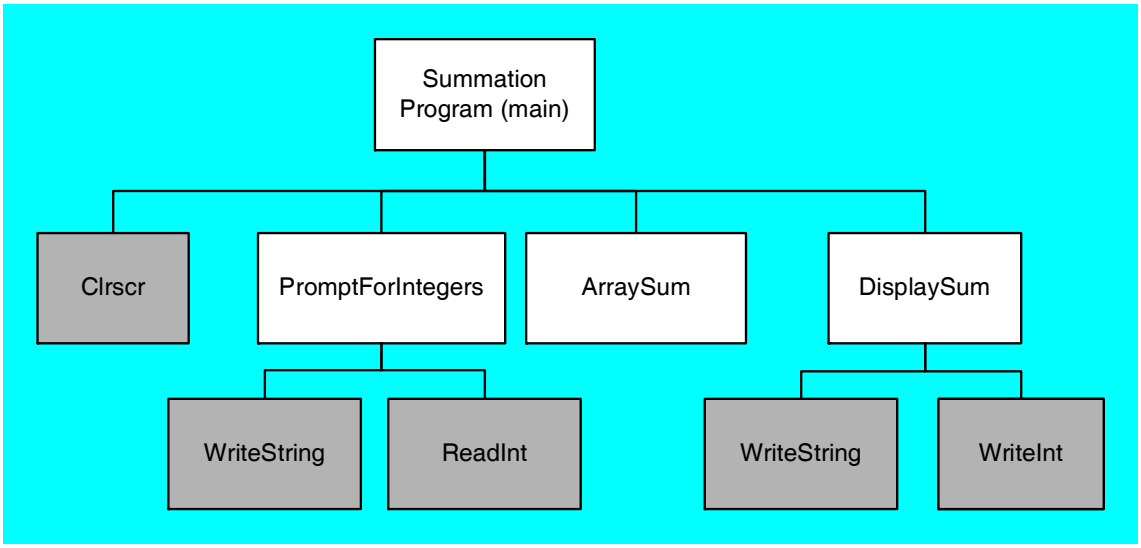
- When not to push a register

```
SumOf PROC    ; sum of three integers
push eax      ; save eax
add eax,ebx   ; calculate the sum of EAX,EBX,ECX
add eax,ecx   ;
pop eax       ; lost the same
ret
SumOf ENDP
```

5.6 Program Design Using Procedures 143

5.6.1 Integer Summation Program (Design) 143

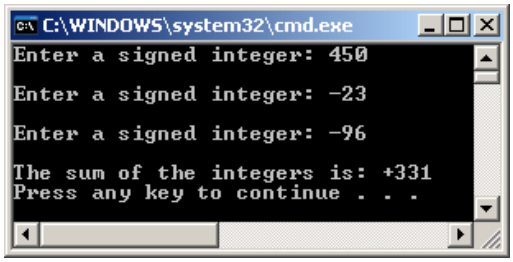
- Top-Down Design (*functional decomposition*) involves the following:
  - Design your program before starting to code
  - Break large tasks into smaller ones
  - Use a hierarchical structure based on procedure calls
  - Test individual procedures separately
- Integer Summation Program
  - *Description:* Write a program that prompts the user for multiple 32-bit integers, stores them in an array, calculates the sum of the array, and displays the sum on the screen.
  - *Main steps:*
    - Prompt user for multiple integers
    - Calculate the sum of the array
    - Display the sum
  - *Structure Chart*



5.6.2 Integer Summation Implementation 145

- *Program code:*

```
TITLE Integer Summation Program          (Sum2.asm)

; This program prompts the user for three integers,
; stores them in an array, calculates the sum of the
; array, and displays the sum.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

INTEGER_COUNT = 3

.data
str1  BYTE  "Enter a signed integer: ",0
str2  BYTE  "The sum of the integers is: ",0
array DWORD  INTEGER_COUNT DUP(?)

.code
main PROC
   call Clrscr
   mov  esi,OFFSET array
   mov  ecx,INTEGER_COUNT
   call PromptForIntegers
   call ArraySum
   call DisplaySum
   exit
main ENDP

;-----------------------------------------------------
PromptForIntegers PROC USES ecx edx esi
;
; Prompts the user for an arbitrary number of integers
; and inserts the integers into an array.
; Receives: ESI points to the array, ECX = array size
; Returns:  nothing
;-----------------------------------------------------
   mov  edx,OFFSET str1  ; "Enter a signed integer"

L1:  call WriteString    ; display string
   call ReadInt          ; read integer into EAX
   call Crlf             ; go to next output line
   mov  [esi],eax        ; store in array
   add  esi,TYPE DWORD   ; next integer
   loop L1

   ret
PromptForIntegers ENDP
```



```
;---------------------------------------------------
ArraySum PROC USES esi ecx
;
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI points to the array, ECX = number
;   of array elements
; Returns:  EAX = sum of the array elements
;---------------------------------------------------
   mov  eax,0            ; set the sum to zero
L1:  add  eax,[esi]      ; add each integer to sum
   add  esi,TYPE DWORD   ; point to next integer
   loop L1               ; repeat for array size
   ret                   ; sum is in EAX
ArraySum ENDP

;---------------------------------------------------
DisplaySum PROC USES edx
;
; Displays the sum on the screen
; Receives: EAX = the sum
; Returns:  nothing
;---------------------------------------------------
   mov  edx,OFFSET str2  ; "The sum of the..."
   call WriteString
   call WriteInt         ; display EAX
   call Crlf
   ret
DisplaySum ENDP

END main
```

5.7 Chapter Summary 147

- This chapter introduce the book's link library to make it easier for you to process input-output in assembly language application.
- Runtime Stack
  - The **runtime stack** is a special array that is used as a temporary holding area for addresses and data.
  - The **ESP** register holds a 32-bit OFFSET into some location on the stack.
  - The stack is called a **LIFO** (last-in, first-out) structure.
  - The **PUSH** instruction **first decrements** the stack pointer and **then copies** a source operand into stack.
  - The **POP** instruction **first copies** the contents of the stack pointed to by ESP into a 16 or 32-bit destination operand and **then increments** ESP.
- Procedure
  - A **procedure** is a named block of code declared using the **PROC** and **ENDP** directives.
  - A procedure's execution ends with the **RET** instruction.
  - The **CALL** instruction executes a procedure by inserting the procedure's address into the instruction pointer register.
  - The **USES** operator, coupled with PROC directive, lets you list all registers modified by a procedure.