Assembly Language
組合語言-資訊工程二年級
Lecture slides(2018 – 2019)

Fu Jen Catholic University, Dept. of Computer Science and Information Engineering –CSIE

資訊工程系-輔仁大學

教授: 周賜福

107學年度第一學期

Assembly Language for x86 Processors 6th Edition

Kip R. Irvine

Chapter 5: Procedures

Slides prepared by the author

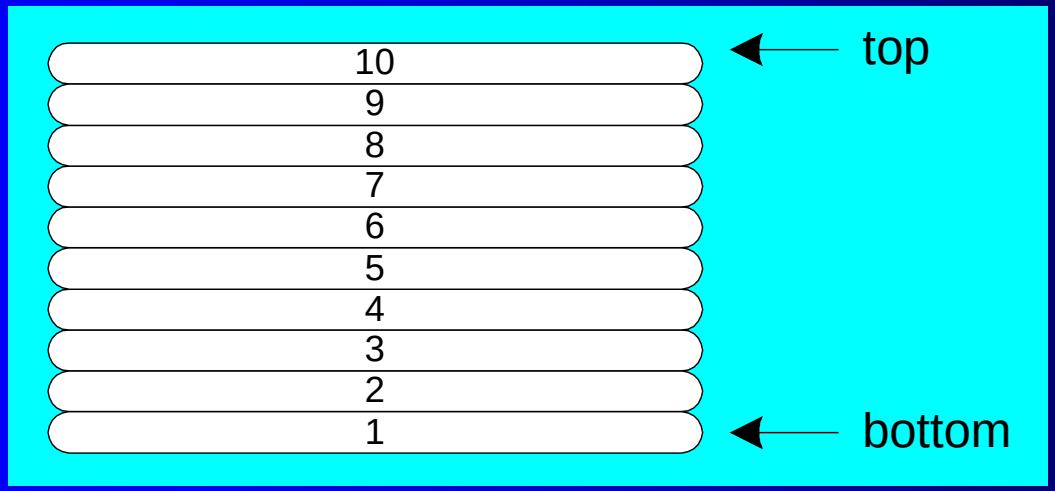
Revision date: 2/15/2010

Chapter Overview

- **Stack Operations**
- **Linking to an External Library**
- The Book's Link Library
- Defining and Using Procedures
- Program Design Using Procedures
- 64-Bit Assembly Programming

5.1 Stack Operation

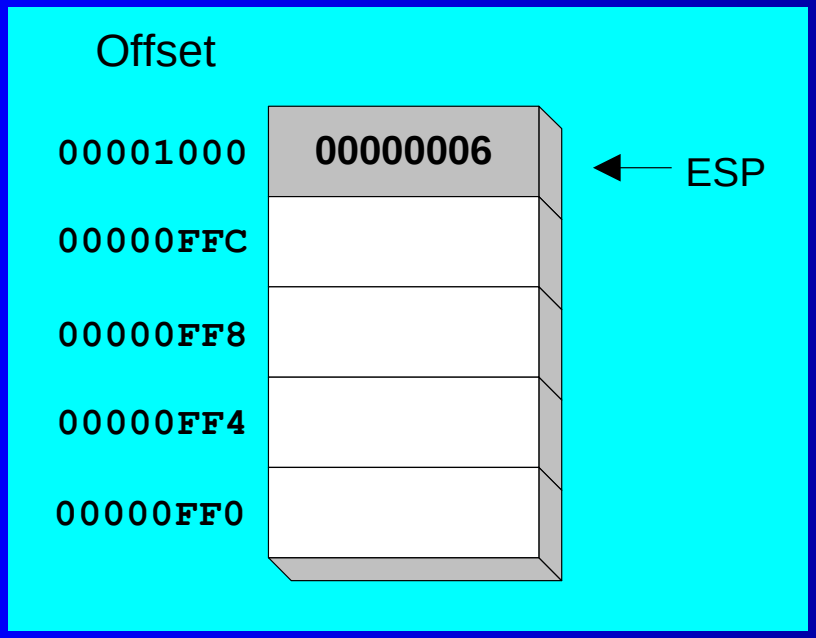
- **Imagine a stack of plates . . .**
  - **plates are only added to the top**
  - **plates are only removed from the top**
  - **LIFO structure (Last In First Out structure)**



In this chapter we concentrate on **runtime stack only**. It is supported directly by hardware in the CPU, and it is an essential part of the mechanism for calling and returning from procedures.

5.1.1 Runtime Stack(32-bit mode)

- **Managed by the CPU, using two registers**
  - **SS (stack segment)**
  - **ESP (Extended Stack Pointer) * (Always points to the last value)**



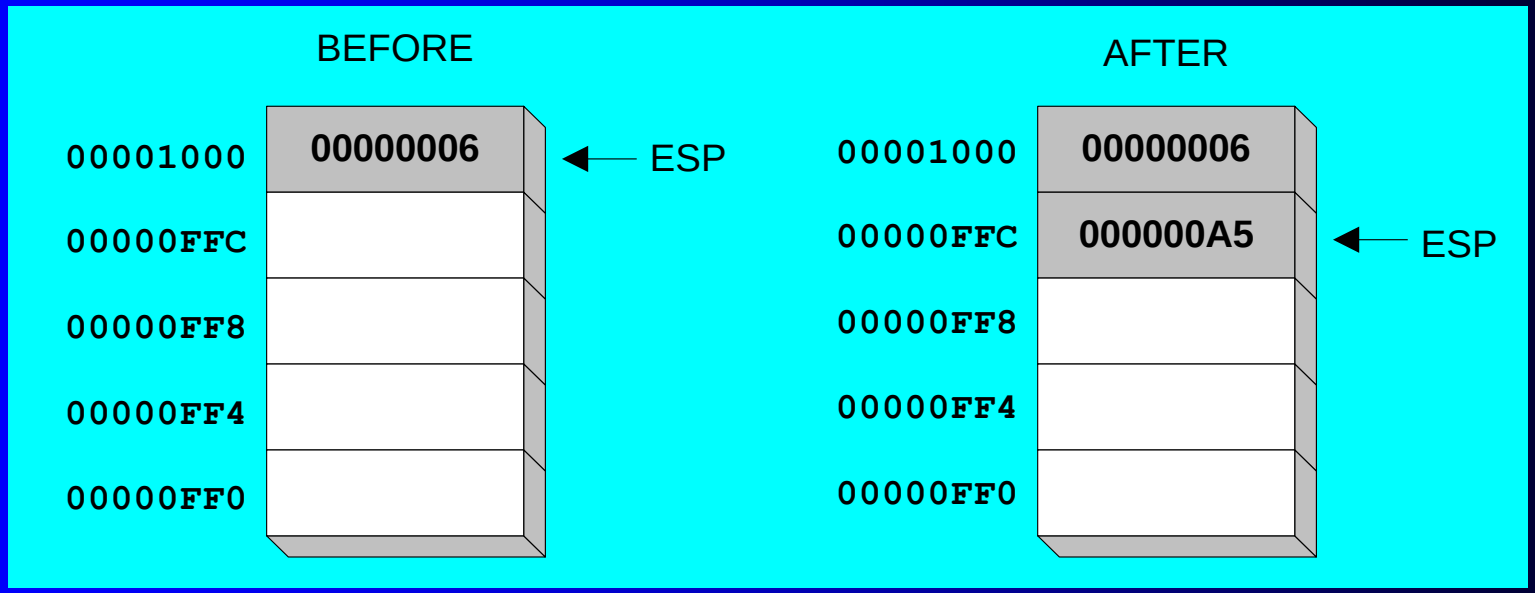
ESP contains 00001000
Value of stack is 00000006

Stack grows downword

* SP in Real-address mode

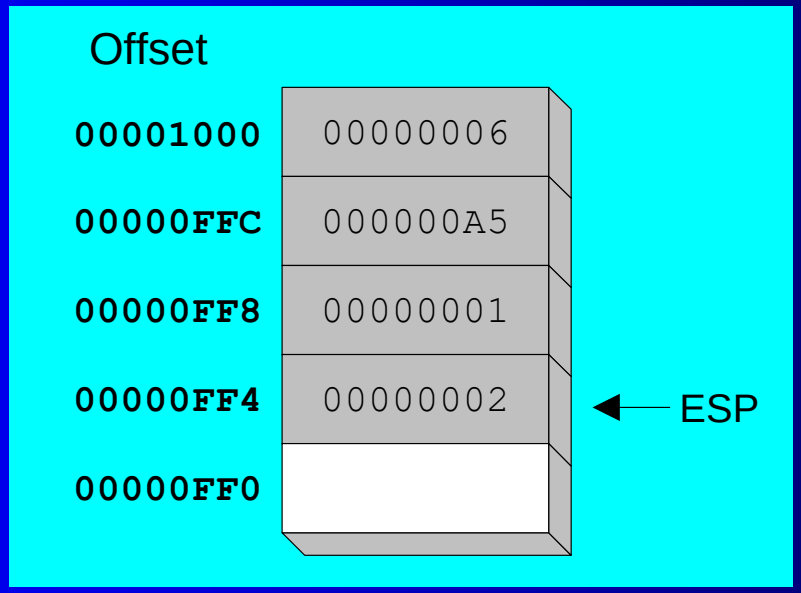
PUSH Operation (1 of 2)

- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.



PUSH Operation (2 of 2)

- Same stack after pushing two more integers:

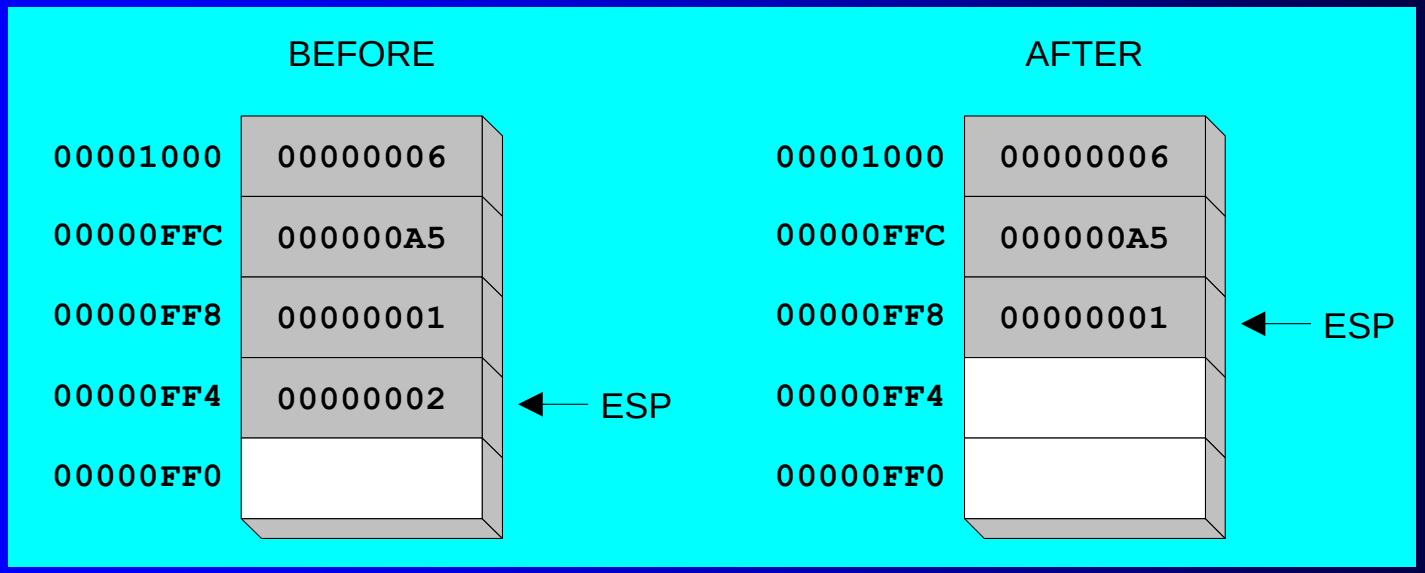


After adding two more values

The stack grows downward. The area below ESP is always available (unless the stack has overflowed).

POP Operation

- Copies value at stack[ESP] into a register or variable.
- Adds *n* to ESP, where *n* is either 2 or 4.
  - value of *n* depends on the attribute of the operand receiving the data



5.1.2 Stack Operations

- Runtime Stack
- PUSH Operation
- POP Operation
- PUSH and POP Instructions
- Using PUSH and POP
- Example: Reversing a String
- Related Instructions

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
```

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**
- **Source code**

- **Q: Why must each character be put in EAX before it is pushed?**

Because only word (16-bit) or doubleword (32-bit) values can be pushed on the stack.

Your turn . . .

- **Using the String Reverse program as a starting point,**
- **#1: Modify the program so the user can input a string containing between 1 and 50 characters.**
- **#2: Modify the program so it inputs a list of 32-bit integers from the user, and then displays the integers in reverse order.**

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**

Why do we need to push the flag?
There are times we need to make a backup copy of the flags so you can restore them to their former values later.

Related information PUSHA, POPA

PUSHA instruction, introduced in 80286 processor, pushes the 16-bit general purpose registers (AX,CX,DXBX,SP,BP,SI,DI)
Similarly, POPA instruction pops these instruction.

IT is basically for 16-bit operands.

Your Turn . . .

- **Write a program that does the following:**
  - **Assigns integer values to EAX, EBX, ECX, EDX, ESI, and EDI**
  - **Uses PUSHAD to push the general-purpose registers on the stack**
  - **Using a loop, your program should pop each integer from the stack and display it on the screen**

4th chapter (slide) Copying a String

The following code copies a string from source to target:

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0)

.code
    mov  esi,0                  ; index register
    mov  ecx,SIZEOF source      ; loop counter
L1:
    mov  al,source[esi]         ; get char from source
    mov  target[esi],al         ; store it in the target
    inc  esi                    ; move to next character
    loop L1                     ; repeat for entire string
```

good use of SIZEOF

SAMPLE PROGRAM USING A STACK

Take a look at the program on page 176

Reverse a string

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

; This program reverses a string.

INCLUDE Irvine32.inc

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

Reverse a string

```
.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
```

Reverse a string

```
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 CrIf

        exit
main ENDP
END main
```

What's Next in 5.2?

- Stack Operations
- Defining and Using Procedures
- The Book's Link Library
- Linking to an External Library
- Program Design Using Procedures

5.2 Defining and using procedures

You have studied high level language programming. You know how useful to divide the program into subroutine.

A complicated problem is usually divided into separate tasks before it can be understood, implemented, and tested effectively.
In assembly language, we typically use the term procedure to mean a subroutine.
In other languages subroutines are called methods or functions.
Assembly language was created long before object-oriented languages. Hence, no formal structure in assembly language. However they must impose their own formal structure on program.

5.2.1 Defining a Procedure

- 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:

```
main PROC
    .
    .

main ENDP
```

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

Documenting Procedures

Suggested documentation for each procedure:

- **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.**

If a procedure is called without its preconditions satisfied, it will probably not produce the expected output.

Example: SumOf Procedure page 179

```
;-----------------------------------------------------
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
```

5.2.2 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

CALL-RET Example (1 of 2)

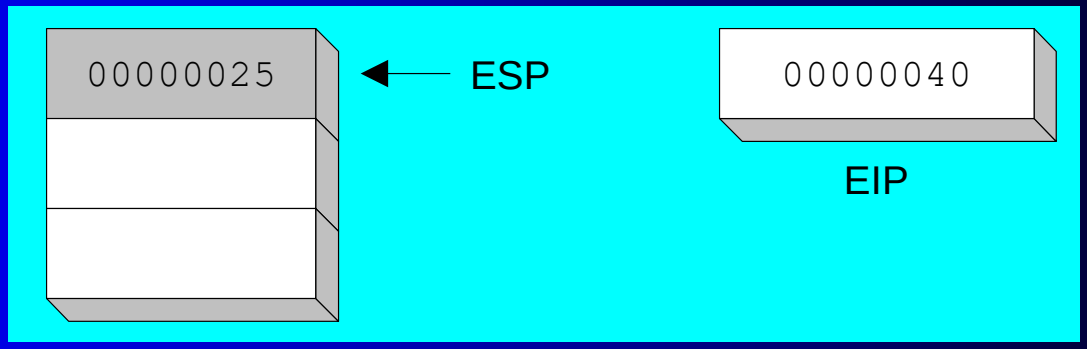
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

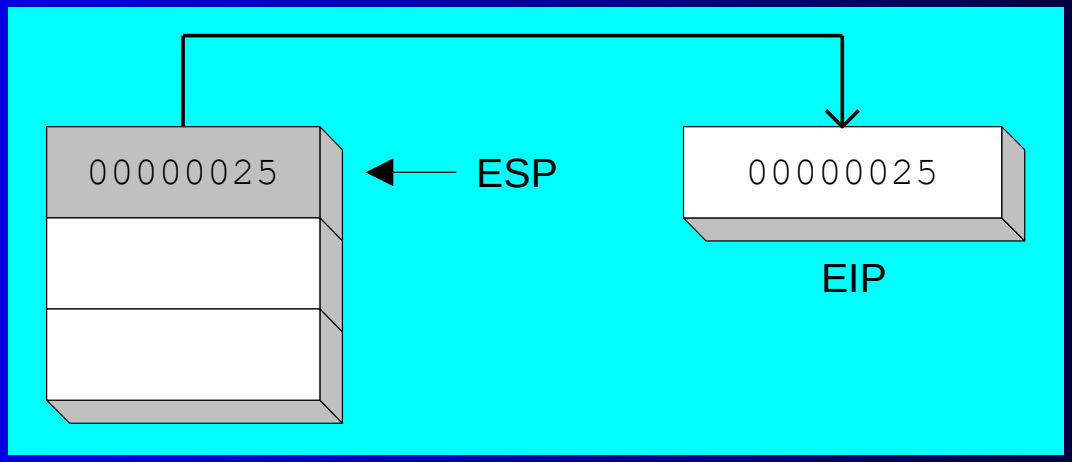
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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

CALL-RET Example (2 of 2)

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



The RET instruction pops 00000025 from the stack into EIP

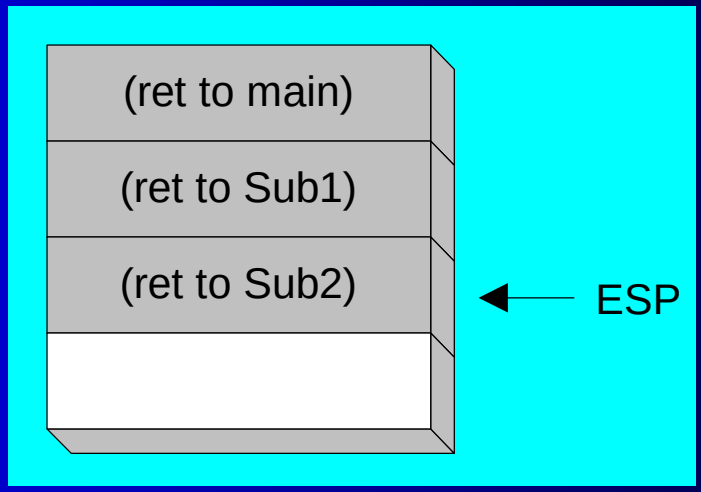


(stack shown before RET executes)

5.2.3 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:



Local and Global Labels

A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

```
main PROC
    jmp L2                  ; error
L1::                        ; global label
    exit
main ENDP

sub2 PROC
L2:                           ; local label
    jmp L1                  ; ok
    ret
sub2 ENDP
```

Procedure Parameters (1 of 3)

- A good procedure might be usable in many different programs
  - but not if it refers to specific variable names
- Parameters help to make procedures flexible because parameter values can change at runtime

Procedure Parameters (2 of 3)

The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

```
ArraySum PROC
    mov esi,0                    ; array index
    mov eax,0                    ; set the sum to zero
    mov ecx,LENGTHOF myarray     ; set number of elements

L1: add eax,myArray[esi]         ; add each integer to sum
    add esi,4                    ; point to next integer
    loop L1                      ; repeat for array size

    mov theSum,eax               ; store the sum
    ret
ArraySum ENDP
```

What if you wanted to calculate the sum of two or three arrays within the same program?

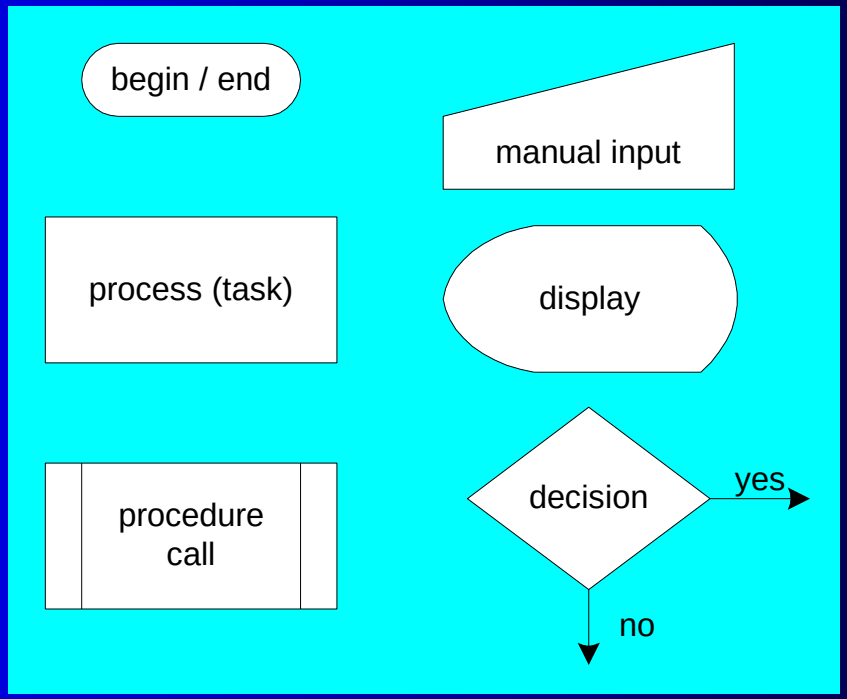
Procedure Parameters (3 of 3)

This version of ArraySum returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
ArraySum PROC
; Receives: ESI points to an array of doublewords,
;   ECX = number of array elements.
; Returns: EAX = sum
;-----------------------------------------------
   mov eax,0                  ; set the sum to zero

L1:add eax,[esi]              ; add each integer to sum
   add esi,4                  ; point to next integer
   loop L1                    ; repeat for array size

   ret
ArraySum ENDP
```

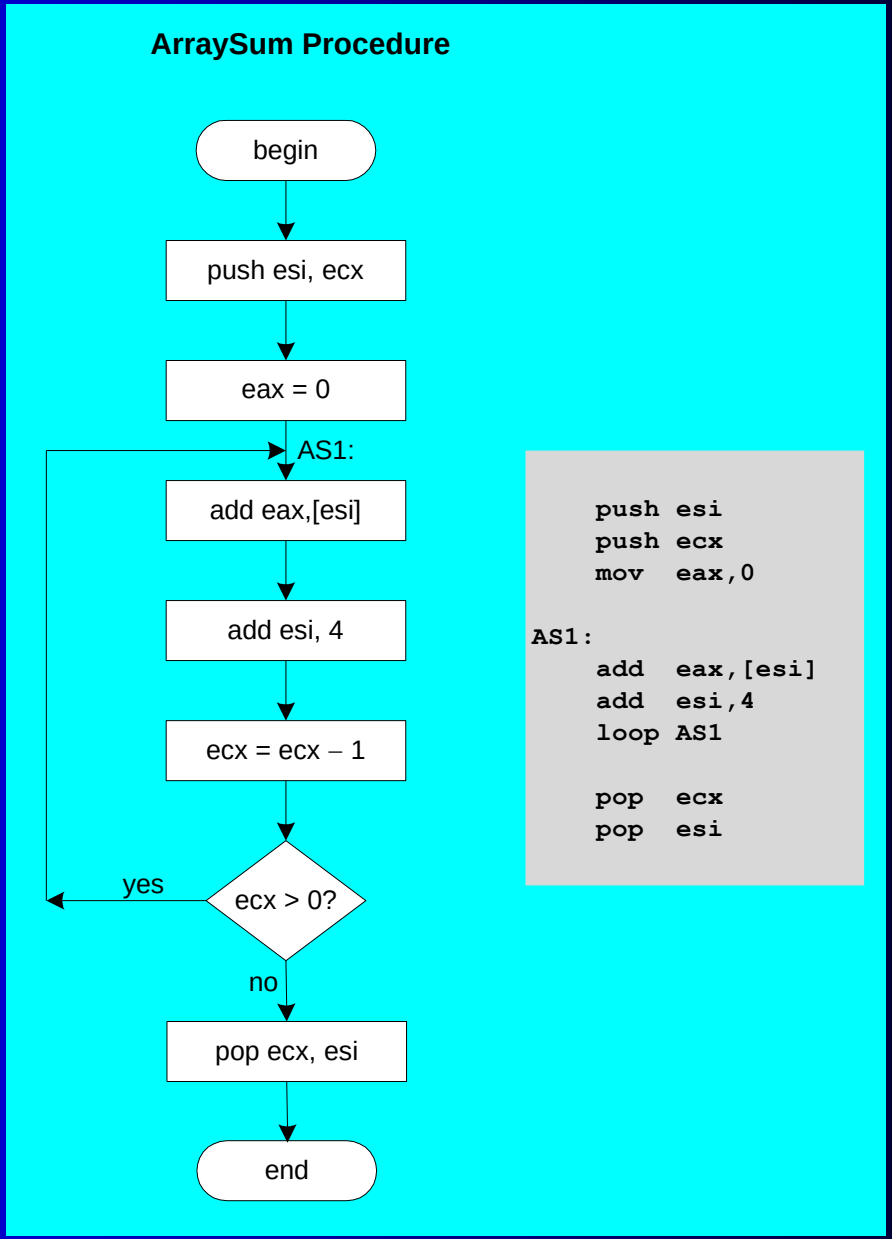
Flowchart Symbols

- **The following symbols are the basic building blocks of flowcharts:**



(Includes two symbols not listed on page 166 of the book.)

Flowchart for the ArraySum Procedure



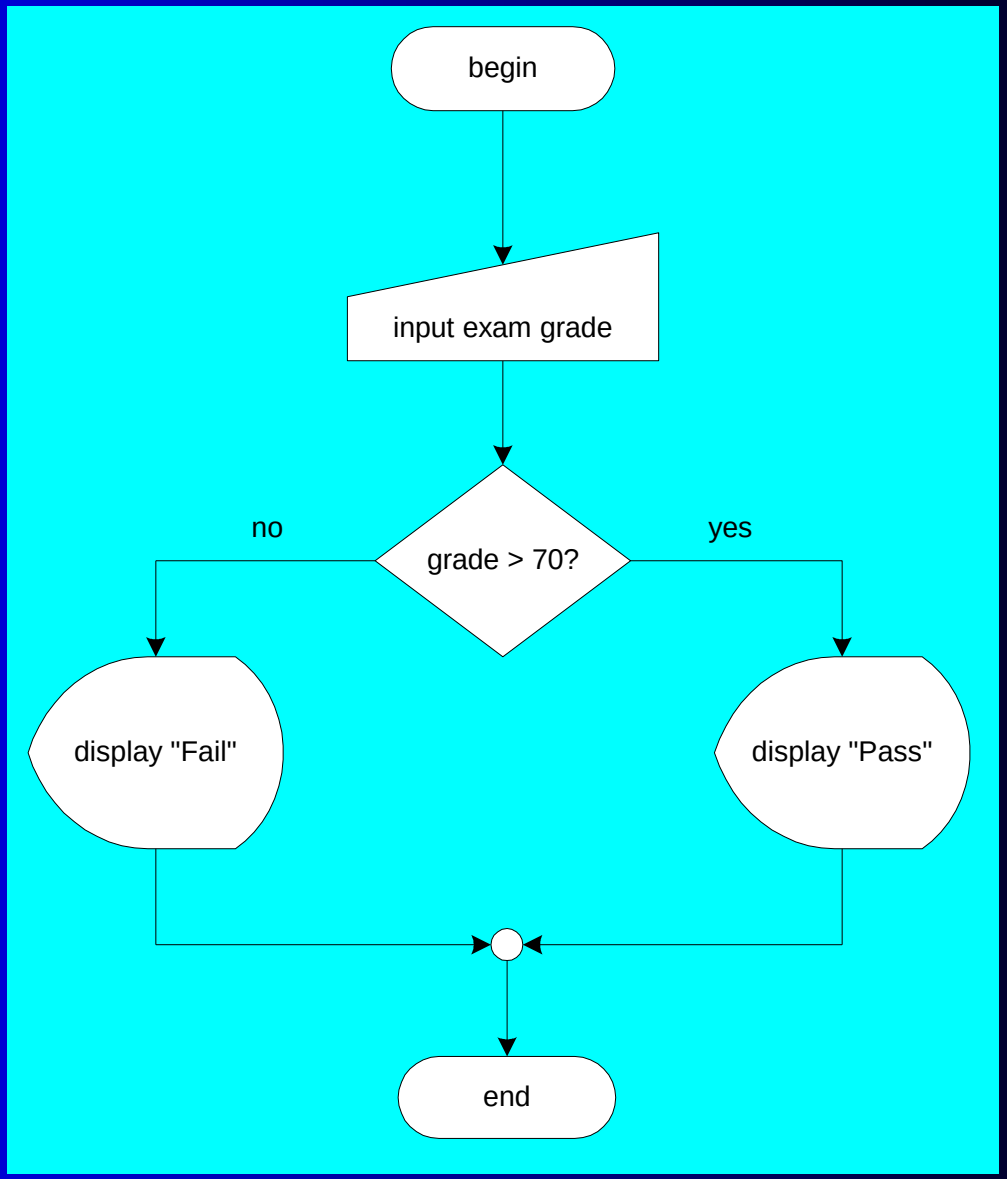

```
    push esi
    push ecx
    mov  eax,0

AS1:
    add  eax,[esi]
    add  esi,4
    loop AS1

    pop  ecx
    pop  esi
```

Your turn . . .

Draw a flowchart that expresses the following pseudocode:

```
input exam grade from the user
if( grade > 70 )
   display "Pass"
else
   display "Fail"
endif
```

. . . (Solution)


begin
input exam grade
grade > 70?
no
yes
display "Fail"
display "Pass"
end

Your turn . . .

- **Modify the flowchart in the previous slide to allow the user to continue to input exam scores until a value of –1 is entered**

5.2.6 Saving and Restoring Registers

- In the Arraysum example, ECX and ESI were pushed on the stack at the beginning of the procedure and popped at the end.
- This action is typical of most procedures that modify registers. Always save and restore registers that are modified by a procedure so the calling program can be sure that none of its own register values will be overwritten.
- The exception to this rule pertains to registers used as return values, usually EAX.
- Do not push and pope them.

USES Operator

- Lists the registers that will be preserved

```
ArraySum PROC USES esi ecx
   mov eax,0                    ; set the sum to zero
   etc.
```

MASM generates the code shown in gold:

```
ArraySum PROC
   push esi
   push ecx
   .
   .
   pop ecx
   pop esi
   ret
ArraySum ENDP
```

When not to push a register

The sum of the three registers is stored in EAX on line (3), but the POP instruction replaces it with the starting value of EAX on line (4):

```
SumOf PROC                  ; sum of three integers
   push eax                 ; 1
   add eax,ebx              ; 2
   add eax,ecx              ; 3
   pop eax                  ; 4
   ret
SumOf ENDP
```

What's Next

- Stack Operations
- Procedures
- Linking to an External Library
- The Book's Link Library
- Defining and Using Procedures
- Program Design Using Procedures

5.3 Linking to an external Library

If you spend time, you can write detailed code for input-output in assembly language. It's a lot like building your own automobile from scratch so that you can drive somewhere. The work is both interesting and time consuming. (Learn in chapter 11 little bit about writing your own input-output.
In this chapter we will learn about how to call procedures form the book's link libraries, named Irvine32.inc and Irvine64.inc.
The complete source code is available at authors website.
The library should be called when program running 32-bit mode. However, 64-bit has limited with some essential display and string procedures.

Link Library Overview

- **A file containing procedures that have been compiled into machine code**
  - **constructed from one or more OBJ files**
- **To build a library, . . .**
  - **start with one or more ASM source files**
  - **assemble each into an OBJ file**
  - **create an empty library file (extension .LIB)**
  - **add the OBJ file(s) to the library file, using the Microsoft LIB utility**

Take a quick look at Irvine32.asm in the \Irvine\Examples\Lib32 folder.

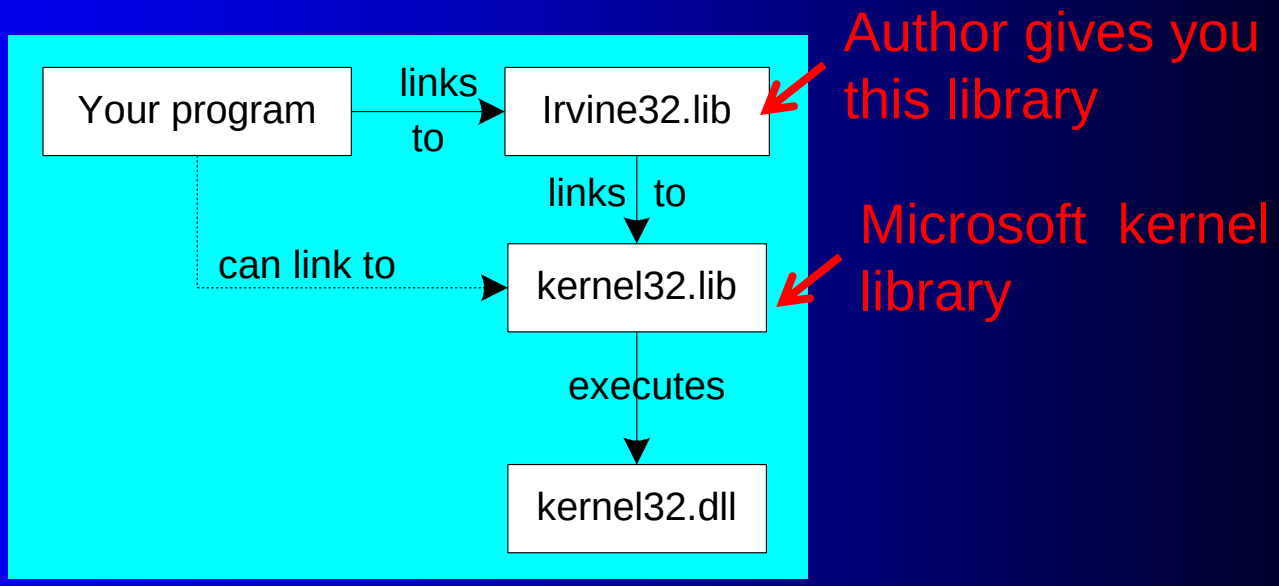
Calling a Library Procedure

- Call a library procedure using the CALL instruction. Some procedures require input arguments. The **INCLUDE** directive copies in the procedure prototypes (declarations).
- The following example displays "1234" on the console:

```
INCLUDE Irvine32.inc
.code
   mov eax,1234h      ; input argument
   call WriteHex      ; show hex number
   call Crlf          ; end of line
```

Linking to a Library

- **Your programs link to Irvine32.lib using the linker command inside a batch file named make32.bat.**
- **Notice the two LIB files: Irvine32.lib, and kernel32.lib**
  - **the latter is part of the Microsoft *Win32 Software Development Kit (SDK)***



What's Next

- Stack Operations
- Linking to an External Library
- The Book's Link Library(Irvine32.inc)
- Defining and Using Procedures
- Program Design Using Procedures

Library Procedures - Overview (1 of 4)

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

GetDateTime – Gets the current date and time from the system

GetMaxXY - Gets number of cols, rows in console window buffer

GetMseconds - Returns milliseconds elapsed since midnight

Library Procedures - Overview (2 of 4)

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, MsgBoxAsk – Display popup message boxes

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

Library Procedures - Overview (3 of 4)

ReadDec - Reads 32-bit unsigned decimal integer from keyboard

ReadFromFile – Reads input disk file into buffer

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

Str_compare – Compares two strings

Str_copy – Copies a source string to a destination string

StrLength – Returns length of a string

Str_trim - Removes unwanted characters from a string.

Library Procedures - Overview (4 of 4)

Str_ucase - Converts a string to uppercase letters.

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

Library Procedures - Overview (5 of 4)

WriteStackFrame - Writes the current procedure's stack frame to the console.

WriteStackFrameName - Writes the current procedure's name and stack frame to the console.

WriteString - Writes null-terminated string to console window

WriteToFile - Writes buffer to output file

WriteWindowsMsg - Displays most recent error message generated by MS-Windows

Irvine Library Help

- **A Windows help file showing:**

- **Irvine Library Procedures**

  Procedure Purpose
  Calling & Return Arguments
  Example of usage

- **Some other information (we will use later)**

IrvineLibHelp.chm

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          ; delays 500 milliseconds
   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 and 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
```

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
```

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

Refer to the Text book for more Example using the Link Library

Page 188 to 189 more example on Link Library is presented

Example programs are given on page 190 ~ 205

What's Next

- Stack Operations
- Linking to an External Library
- The Book's Link Library
- Defining and Using Procedures
- Program Design Using Procedures

Defining and Using Procedures

- Creating Procedures
- Documenting Procedures
- Example: SumOf Procedure
- CALL and RET Instructions
- Nested Procedure Calls
- Local and Global Labels
- Procedure Parameters
- Flowchart Symbols
- USES Operator

What's Next

- Linking to an External Library
- The Book's Link Library
- Stack Operations
- Defining and Using Procedures
- **Program Design Using Procedures**

Program Design Using Procedures

- 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 (1 of 4)

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.**

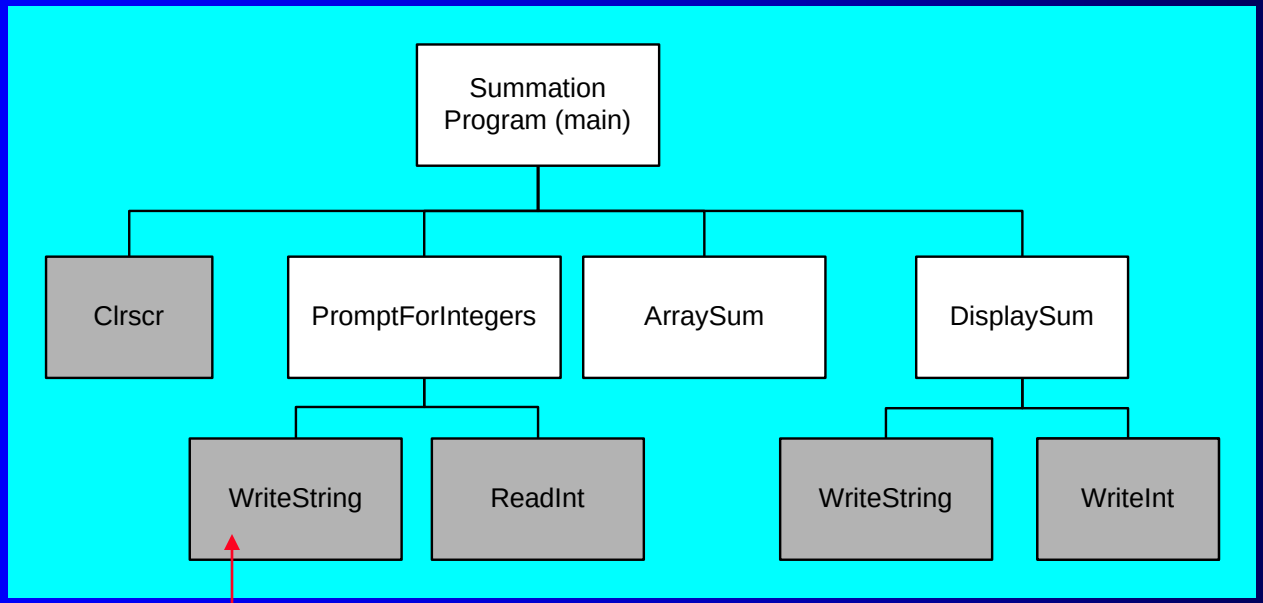
Main steps:

- **Prompt user for multiple integers**
- **Calculate the sum of the array**
- **Display the sum**

Procedure Design (2 of 4)

```
Main
    Clrscr                      ; clear screen
    PromptForIntegers
        WriteString             ; display string
        ReadInt                 ; input integer
    ArraySum                    ; sum the integers
    DisplaySum
        WriteString             ; display string
        WriteInt                ; display integer
```

Structure Chart (3 of 4)



gray indicates library procedure

- View the stub program
- View the final program

Sample Output (4 of 4)

```
Enter a signed integer: 550
Enter a signed integer: -23
Enter a signed integer: -96
The sum of the integers is: +431
```

Library Test #2 Random Integers(1/3)

Let's take look at the second library test program that demonstrates random-number-generation capabilities of the link library, and introduces the CALL instruction.

```
; Link Library Test #2    (TestLib2.asm)
; Testing the Irvine32 Library procedures.
INCLUDE Irvine32.inc
TAB = 9                    ; ASCII code for Tab
.code
main PROC
        call    Randomize      ; init random generator
        call    Rand1          ; The programmer must write this
        call    Rand2          ; The programmer must write this
main ENDP
```

Library Test #2 Random Integers(1/3)

First procedure is written here.

```
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
```

Library Test #2 Random Integers(1/3)

Second procedure is written here.

```
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
```

What's Next

- Stack Operations
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library
- **64-Bit Assembly Programming**

64-Bit Assembly Programming

- The Irvine64 Library
- Calling 64-Bit Subroutines
- The x64 Calling Convention

The Irvine64 Library

- Crlf: Writes an end-of-line sequence to the console.
- Random64: Generates a 64-bit pseudorandom integer.
- Randomize: Seeds the random number generator with a unique value.
- ReadInt64: Reads a 64-bit signed integer from the keyboard.
- ReadString: Reads a string from the keyboard.
- Str_compare: Compares two strings in the same way as the CMP instruction.
- Str_copy: Copies a source string to a target location.
- Str_length: Returns the length of a null-terminated string in RAX.
- WriteInt64: Displays the contents in the RAX register as a 64-bit signed decimal integer.

The Irvine64 Library (cont'd)

- WriteHex64: Displays the contents of the RAX register as a 64-bit hexadecimal integer.
- WriteHexB: Displays the contents of the RAX register as an 8-bit hexadecimal integer .
- WriteString: Displays a null-terminated ASCII string.

Calling 64-Bit Subroutines

- Place the first four parameters in registers
- Add PROTO directives at the top of your program
  - examples:

```
ExitProcess PROTO    ; located in the Windows API
WriteHex64 PROTO     ; located in the Irvine64 library
```

The x64 Calling Convention

- Must use this with the 64-bit Windows API
- CALl instruction subtracts 8 from RSP
- First four parameters must be placed in RCX, RDX, R8, and R9
- Caller must allocate at least 32 bytes of shadow space on the stack
- When calling a subroutine, the stack pointer must be aligned on a 16-byte boundary.

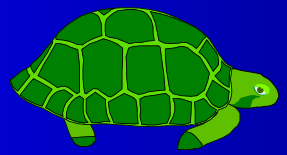
See the CallProc_64.asm example program.

Summary

- Procedure – named block of executable code
- Runtime stack – LIFO structure
  - holds return addresses, parameters, local variables
  - PUSH – add value to stack
  - POP – remove value from stack
- Use the Irvine32 library for all standard I/O and data conversion
  - Want to learn more? Study the library source code in the c:\Irvine\Examples\Lib32 folder

Summary

- Procedure – named block of executable code
- Runtime stack – LIFO structure
  - holds return addresses, parameters, local variables
  - PUSH – add value to stack
  - POP – remove value from stack
- Use the Irvine32 library for all standard I/O and data conversion
  - Want to learn more? Study the library source code in the c:\Irvine\Examples\Lib32 folder



End of Chapter 5 slides.