Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 5: Procedures

（过程）

Chapter Overview

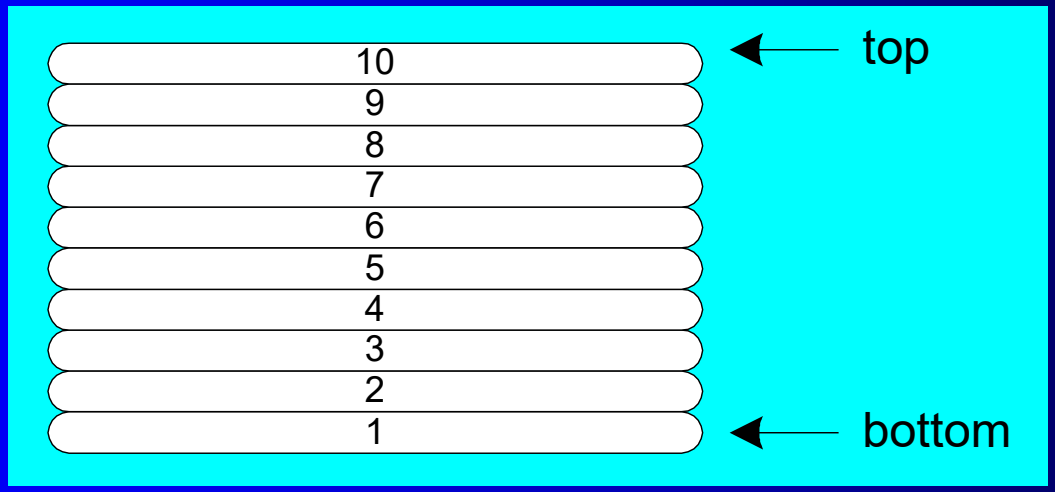
- Stack Operations
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library
- 64-Bit Assembly Programming

Stack Operations
堆栈

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

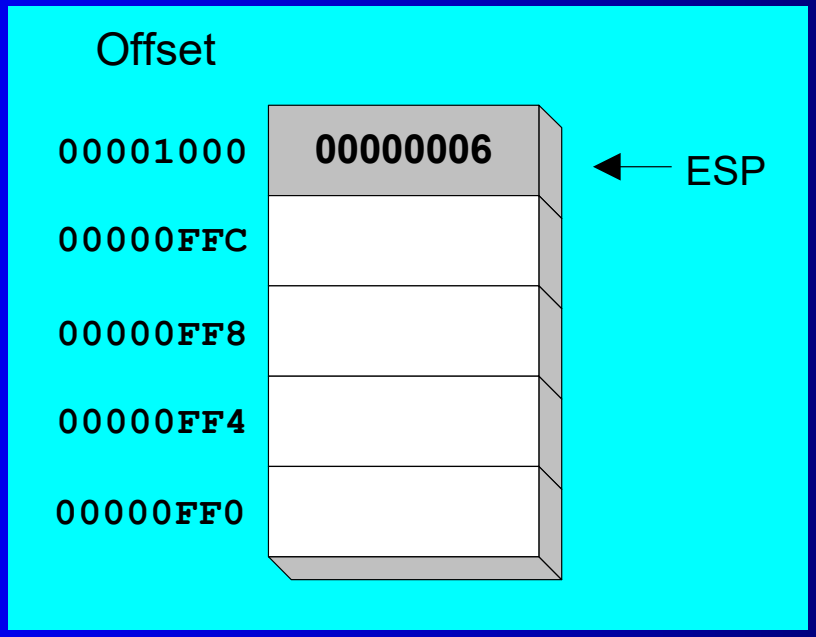
Runtime Stack

- Imagine a stack of plates . . .
  - plates are only added to the top
  - plates are only removed from the top
  - LIFO structure



Runtime Stack

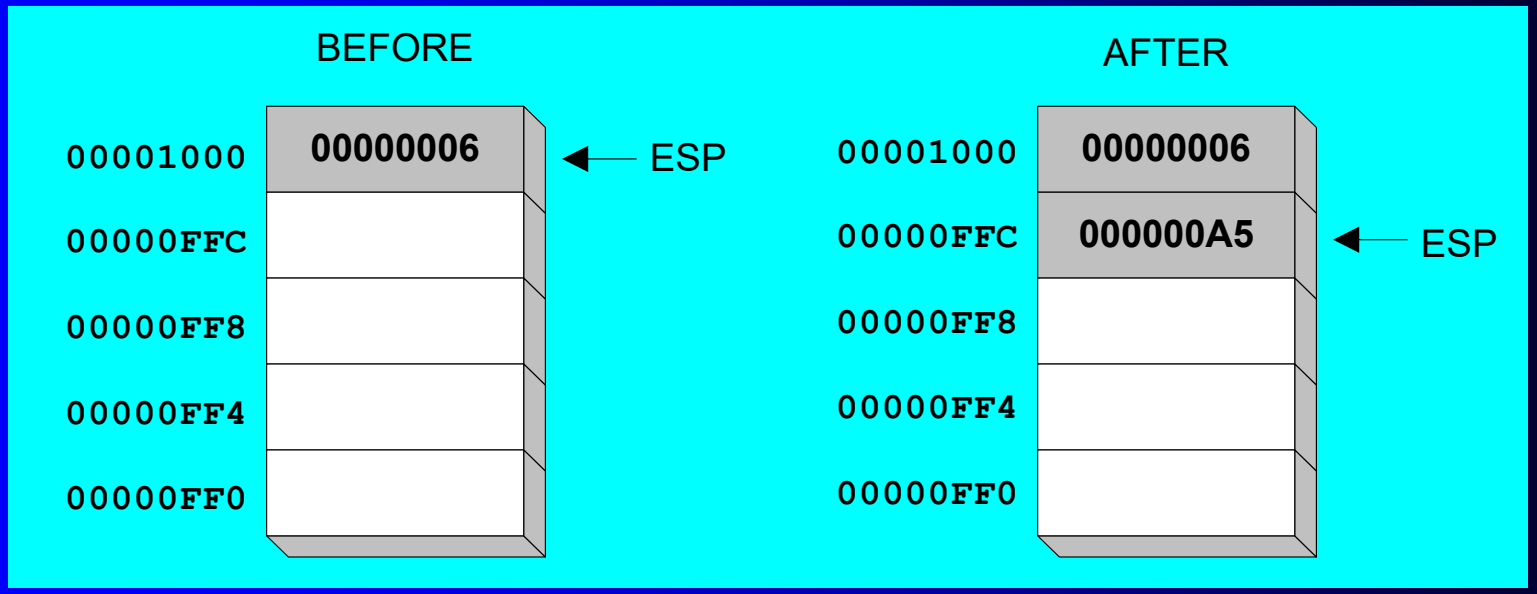
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *



* SP in Real-address mode

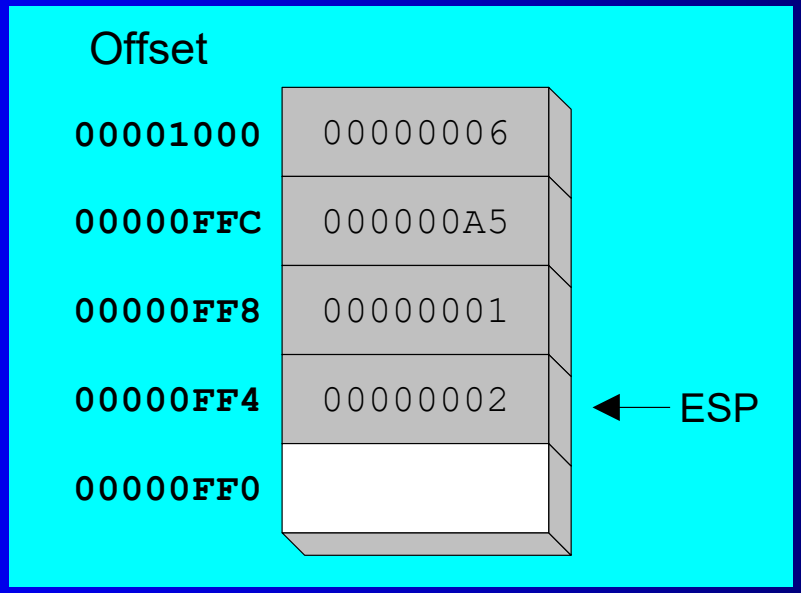
PUSH Operation (1 of 2)

- 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)

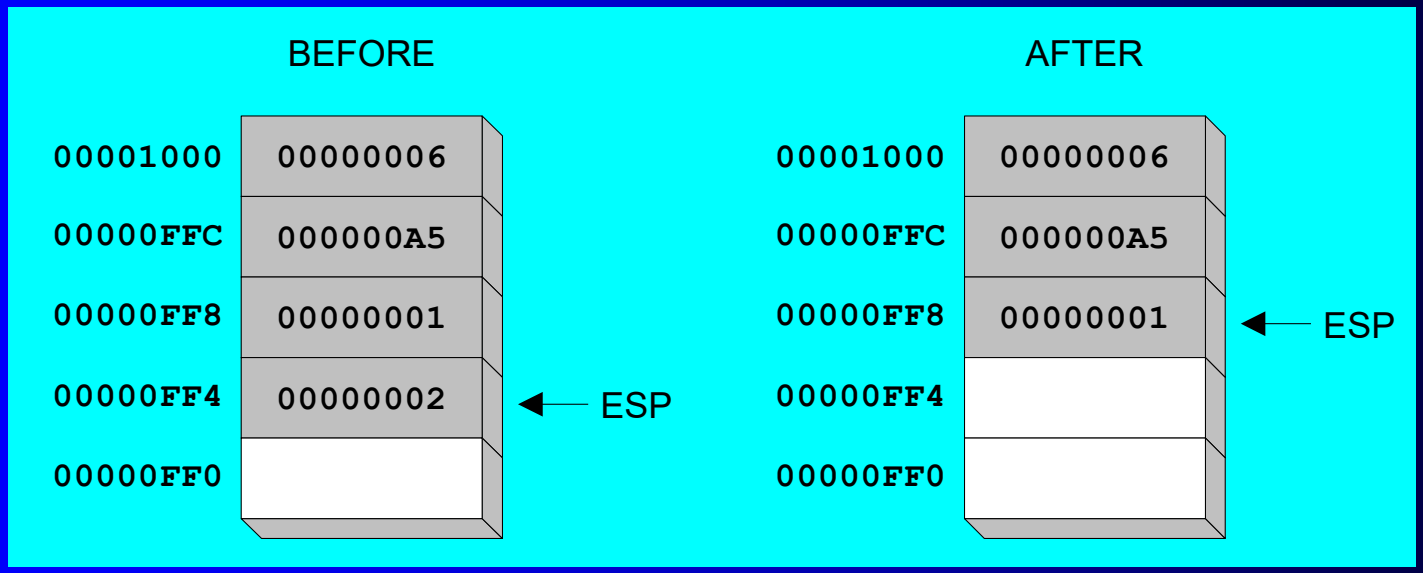
- Same stack after pushing two more integers:



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



PUSH and POP Instructions

- PUSH syntax:
  - PUSH *r/m16*
  - PUSH *r/m32*
  - PUSH *imm32*
- POP syntax:
  - POP *r/m16*
  - POP *r/m32*

```
push esi                          ;
push ecx
push ebx

mov  esi,OFFSET dwordVal          ;
mov  ecx,LENGTHOF dwordVal
mov  ebx,TYPE dwordVal
call DumpMem

pop  ebx                          ;
pop  ecx
pop  esi
```

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                ; 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.

```
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
```

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

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

What's Next

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

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

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

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

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

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)

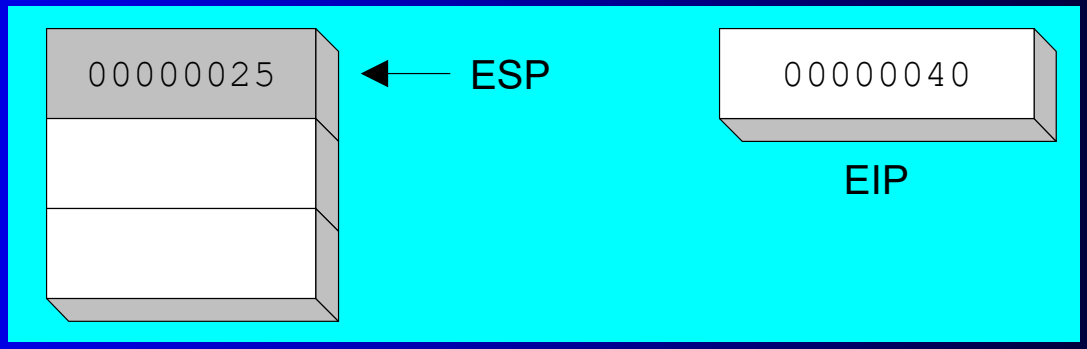
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

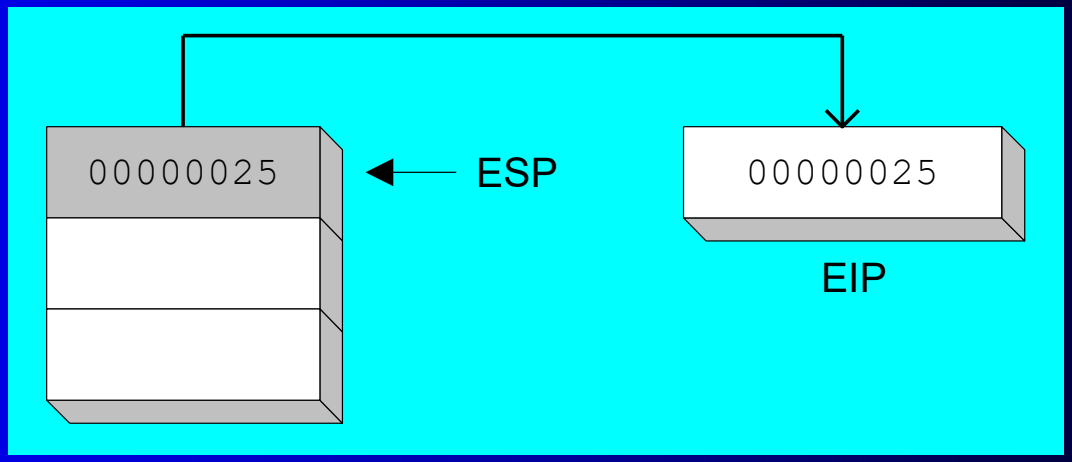
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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

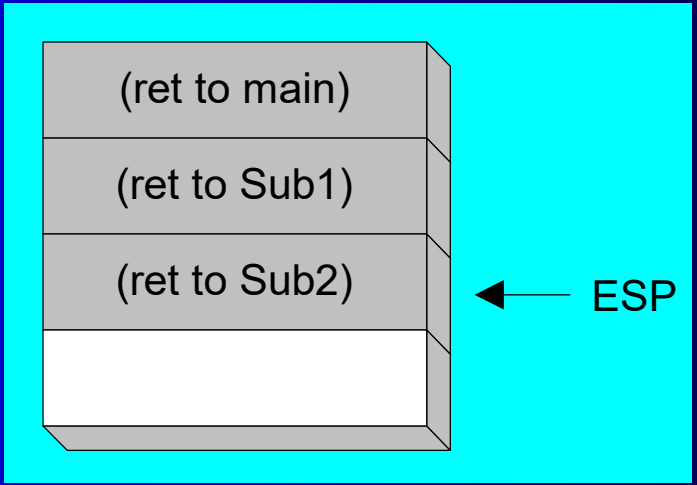


(stack shown before RET executes)

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
                                    ; double ::
    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
```

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

Linking to an External Library

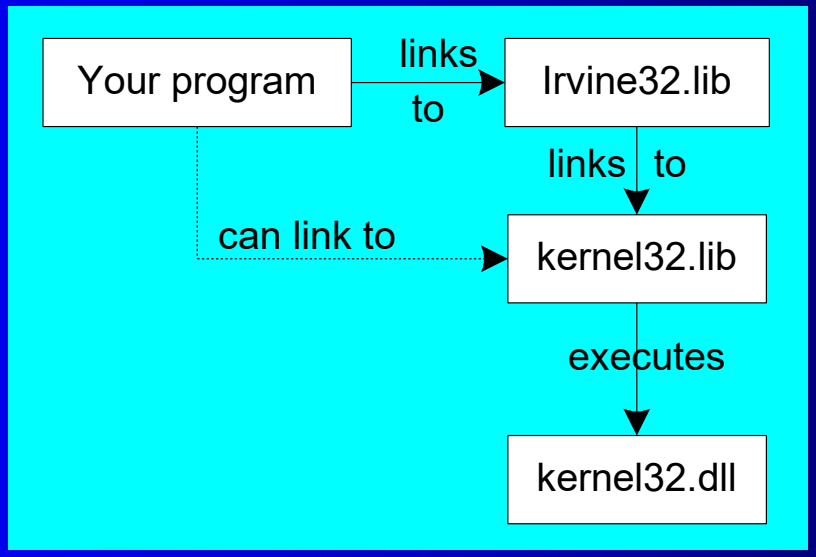
- What is a Link Library?
- How the Linker Works

What is a Link Library?

- 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

How The Linker Works

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

Calling Irvine32 Library Procedures

- Call each 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
```

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 stdin, terminated by [Enter]

SetTextColor - Sets foreground/background colors of all subsequent text output to the console

Str_compare – Compares two strings

Str_copy – Copies a source string to a destination string

Str_length – Returns the length of a string in EAX

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

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 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

L1: mov  eax,100
    call RandomRange
    call WriteInt
    call Crlf
    loop L1
```

```
.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
   ……
```

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

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

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

What's Next

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

- call WriteString
- Call Writestring
- No case sensitive, but may be red line in editor.

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

- Write a program that generates and displays 20 random strings, each consisting of 10 capital letters

- Description: Create a procedure that generates a random string of length L, containing all capital letters.
- When calling the procedure, pass the value of L in EAX, and pass a pointer to an array of byte that will hold the random string. Write a test program that calls your procedure 20 times and displays the strings in the console window

Exercise

- 5.1
- 5.3
- 5.5
- 5.6