PROCEDURES

CS 16: Assembly Language Programming for the IBM PC and Compatibles
OBJECTIVES

- Begin with getting to know stack operations
- Learn how to define and use procedures
- See how we link to an external library
- Figure out how to design a program using procedures
- Dive deeper into the Irvine32 library
- Look at its application to 64-Bit Assembly Programming
STACK OPERATIONS

- Runtime stack
- PUSH operation
- POP operation
- PUSH and POP instructions
- Using PUSH and POP
- Example: Reversing a string
- Related instructions
Imagine a stack of plates . . .

- Plates are only added to the top
- Plates are only removed from the top
- Last-In, First-Out (LIFO) structure
Managed by the CPU, using two registers
- SS (stack segment)
- ESP (stack pointer; SP in real-address mode)
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)**

- Same stack after pushing two more integers
  - The stack grows downward
  - The area below the 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

```
BEFORE
00000006
000000A5
00000001
00000002
ESP

00001000
00000FFC
00000FF8
00000FF4
00000FF0

AFTER
00000006
000000A5
00000001
00000002
ESP

00001000
00000FFC
00000FF8
00000FF4
00000FF0
```
PUSH AND POP INSTRUCTIONS

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

- **POP syntax**
  - POP r/m16
  - POP r/m32
Save and restore registers when they contain important values

PUSH and POP instructions occur in the opposite order

```assembly
call DumpMem
mov ebx,TYPE dwordVal
mov ecx,LENGTHOF dwordVal
mov esi,OFFSET dwordVal
push esi ; push registers
push ecx
push ebx
pop esi ; restore registers
pop ecx
pop ebx
```
When creating a nested loop, push the outer loop counter before entering the inner loop

```assembly
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
- 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
Using the String Reverse program as a starting point...

- Modify the program so the user can input a string containing between 1 and 50 characters
- Modify the program so it inputs a list of 32-bit integers from the user and then displays the integers in reverse order
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
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

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

- 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)

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

- The RET instruction pops 00000025 from the stack into EIP
A local label is visible only to statements inside the same procedure
A global label is visible everywhere

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

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

```
ArraySum PROC
    push esi
    push ecx
    .
    .
    pop ecx
    pop esi
    ret
ArraySum ENDP
```
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 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
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)
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
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
Main
  Clrscr ; clear screen
  PromptForIntegers
    WriteString ; display string
    ReadInt ; input integer
  ArraySum ; sum the integers
  DisplaySum
    WriteString ; display string
    WriteInt ; display integer
Items in GRAY indicate library procedures
Enter a signed integer: 550
Enter a signed integer: -23
Enter a signed integer: -96
The sum of the integers is: +431
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

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

- 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
LIBRARY PROCEDURES (2)

- **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
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
Library Procedures (4)

- 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 (5)

- **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]
LIBRARY PROCEDURES (6)

- **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 (7)

- **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.
Library Procedures (8)

- 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 (9)**

- **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.
Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags

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

Sample output:

```plaintext
EAX=00000613 EBX=00000000 ECX=00000FF EDX=00000000
ESI=00000000 EDI=0000100 EBP=0000091E ESP=00000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```
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
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
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
```
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
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
```
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.
64-BIT 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
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
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