Chapter 5: Procedures

Kip R. Irvine

Chapter Overview

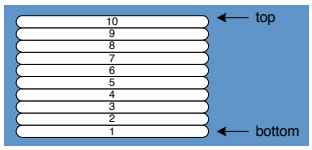
- Stack Operations
- Defining and Using Procedures
- Program Design Using Procedures
- Linking to an External Library
- The IO.h Library
- 64-Bit Assembly Programming

Stack Operations

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

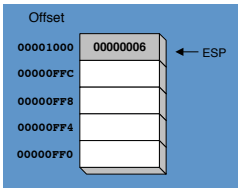
Runtime Stack

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



Runtime Stack

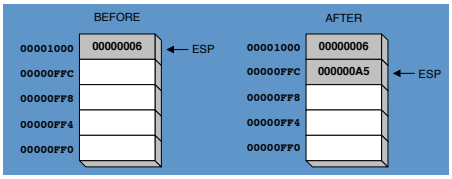
- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *



* SP in Real-address mode

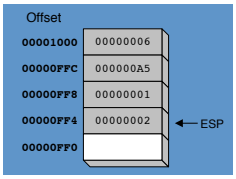
PUSH Operation (1 of 2)

- 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)

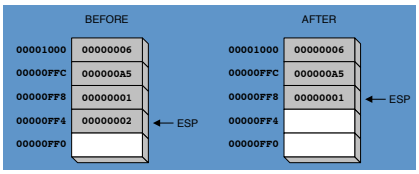
- Same stack after pushing two more integers:



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*

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
- Start at the beginning of the string, push each character in the string on the stack
- Start at the beginning of the string, pop from the stack (in reverse order), insert each character back into the string

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

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

Reversing a String

```
...
.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
...
```


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

What's Next

- Stack Operations
- **Defining and Using Procedures**
- Program Design Using Procedures
- Linking to an External Library
- The IO.h 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 Example
- USES Operator

Creating Procedures

- Large problems can be divided into smaller tasks to make them more manageable
- Procedure is a named block of statements that ends in a return statement
- Declared using PROC and ENDP directives
- Must be assigned a name (valid identifier)
- 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)

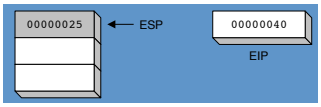
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

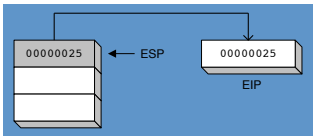
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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

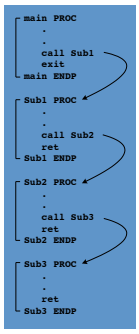


The RET instruction pops 00000025 from the stack into EIP

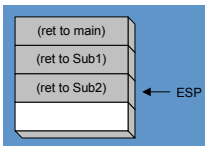


(stack shown before RET executes)

Nested Procedure Calls



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. Global label identified by double colon (::)

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

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

Not good to jump or loop outside the current procedure, could corrupt runtime stack.

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
```

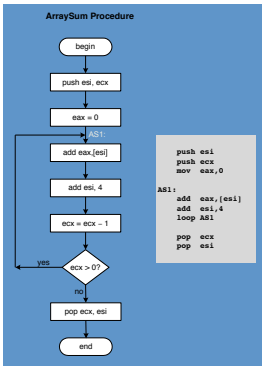
What if you wanted to calculate the sum of two or three arrays within the same program?

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 for the ArraySum Procedure



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 red:

```
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
- **Program Design Using Procedures**
- Linking to an External Library
- The IO.h Library
- 64-Bit Assembly Programming

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.

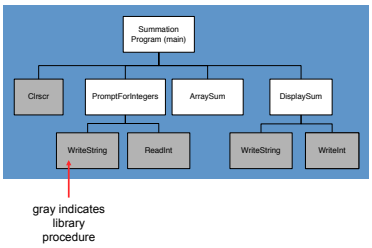
Main steps:

- 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)



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
```

What's Next

- Stack Operations
- Defining and Using Procedures
- Program Design Using Procedures
- **Linking to an External Library**
- The IO.h Library
- 64-Bit Assembly Programming

Linking to an External Library

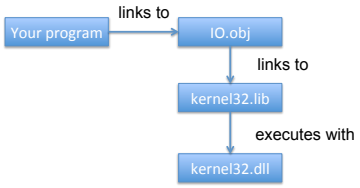
- What is a Link Library?
- How the Linker Works

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 IO.obj and kernel32.lib using the linker command inside a batch file named lab1.mak
- Notice the two files: IO.lib, and kernel32.lib
  - kernel32 is part of the Microsoft Win32 Software Development Kit (SDK), which is a dynamic link library



What's Next

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

IO.h Library

- In order to write useful programs, we need to be able to handle console input and output
  - IO.h Library support these functions
- For example, a typical numeric input function
  - Accepts a string of character codes representing a number
  - Converts the characters to a 2's complement in a register
  - Stores the value in a memory location associated with some variable name
- A numeric output function
  - Starts with a 2's complement number in a register
  - Converts value to a string of characters that represent the number
  - Outputs the string

Calling IO.h Library Procedures

- Call each procedure using macro directive. Some procedures require input arguments. The include directive copies in the procedure prototypes (declarations).
- The following example displays "Hello" on the console:

```
…
include IO.h

.DATA

    szHello BYTE "Hello",0

.CODE
_start:
    output szHello          ; output string
…
```


output Macro (print ASCII characters)

- The output macro displays a string of characters starting at *source* location in the data segment:

  output *source*
  - where *source* is the beginning of a null-terminated string in memory
  - For example:

```
.DATA ; Data segment
    TXTPROMPT BYTE "HELLO", 0
    ENDL BYTE 13, 10, 0
..
.CODE ; Code segment
..
    output TXTPROMPT ; print HELLO
    output ENDL ; print new line
..
```

  - Characters starting at *source* address are displayed until a **null character** is reached. The null character terminates the output
  - Flags or registers affected: *none*

input Macro (read ASCII characters)

- Inputs a string of characters from the keyboard, It has two parameters, *destination* and *length*:

  input *destination*, *length*
  - where *destination* operand references a string of bytes in the data segment and *length* operand references the number of bytes reserved in the *destination* string.
  - For example:

```
.DATA ; Begin initialized data segment
    buffer BYTE 12 DUP (?)
…
.CODE ; Code segment
…
    input buffer, 12   ; Read zero to 10 ASCII characters
…
```


input Macro (read ASCII characters)

- Important: The destination string should be *at least two bytes longer* than the actual number of characters to be entered
- This allows for the operating system to add *carriage return* and *line feed* characters when user presses the **Enter** key
- The input macro automatically replaces the *carriage return* character by a **null byte**
- The result is a null-terminated string stored at the *destination*
- The input macro updates memory at the specified *destination*
- Flags or registers affected: *none*

szlen Macro (string length)

- The szlen macro calculates the length of null-terminated string in memory:

  szlen *source*
  - where *source* is the beginning of a null-terminated string in memory
  - For example:

```
.DATA ; Begin initialized data segment buffer
BYTE 12 DUP (?)
…
.CODE ; Code segment
…
input buffer, 12 ; Read zero to 10 ASCII characters
szlen buffer ; Calculate user input length (result in EAX)
…
```

  - The length is returned in EAX.
  - Flags affected: *none*

dtoa Macro (DWORD to ASCII)

- The dtoa converts 32-bit signed integer (doubleword) to **eleven-byte-long** ASCII string at *destination*

  dtoa *destination*, *source*
  - Where *destination* operand is a string of exactly **11** ASCII characters in the data segment and *source* operand is normally a register or memory operand
  - For example:

```
.DATA ; Begin initialized data segment
    minus_one DWORD 0FFFFFFFFh
    dtoa_buffer BYTE 11 DUP (?)
…
.CODE ; Code segment
…
    dtoa dtoa_buffer, eax ; Convert EAX value to string
    output dtoa_buffer ; Print result
…
    dtoa dtoa_buffer, [minus_one]
    output dtoa_buffer ; Print result ..
```

dtoa Macro (DWORD to ASCII)

- The name dtoa stands for *double to ASCII*
- The result represents signed integer in the decimal number system
- The *destination* is always **11**-byte area of storage in the data segment reserved with a BYTE directive
- The resulting string of characters has leading blanks if decimal number is shorter than **11** characters:
  - If the number is negative, a minus sign is immediately preceding the digits
  - Since the decimal range for a dword-length 2's complement number is **-2,147,483,648** to **2,147,483,647**
  - there is no danger of generating too many characters to fit in an **11**-byte-long field
  - A positive number will always have at least one leading blank
- Flags or registers affected: *none*.

atod Macro (ASCII to DWORD)

- The atod converts an ASCII string at *source* to 32-bit signed integer number
  - atod *source*
  - where *source* is a null-terminated string in the following format:
  - optional leading blanks (space characters, ASCII codes 20h), optional **+** or **-** sign, followed by digits
  - Resulting 2's complement number is put in EAX
  - For example:

```
.DATA ; Begin initialized data segment
    buffer BYTE 12 DUP (?)
…
.CODE ; Code segment
…
    input buffer, 12 ; Read zero to 10 ASCII characters
    atod buffer ; Convert string to 2's complement number
    jno L1 ; Check the overflow flag
    ; Handle input error:
…
L1:
    ; Success: result of conversion is in EAX
```


atod Macro (ASCII to DWORD)

- The offset of the terminating non-digit character (usually NULL, used to stop the conversion) is placed in ESI
- **If no input error detected**,
  - The **overflow flag** OF is cleared
  - **Other flags** are set accordingly to the result in EAX:
    - SF is 1 if the number is negative, and 0 otherwise
    - ZF is 1 if the number is 0, and 0 if the number is nonzero
    - PF reflects the parity of the number returned in EAX
    - In addition, CF is 0 and DF is unchanged
- **If input error is detected**,
  - The overflow flag OF is set to **1**
  - EAX register contains zero
- Input error occurs if the *source* has no digits or is out of the range **-2,147,483,648** to **2,147,483,647**
- If atod is applied to a string that comes from some source other than input, the programmer must ensure that the string has some trailing non-digit character to prevent atod from scanning too far

atoi and itoa Macros

- The atoi (ASCII to integer), and itoa (integer to ASCII)
  - macros are the **word-length** (16-bit) versions of atod and dtoa
- The atoi macro scans a string of characters and produces the corresponding word-length 2's complement value in AX
- The itoa macro takes the 2's complement value stored in a word-length source and produces a string of *exactly six characters* representing this value in decimal format
- The macros are useful when dealing with values in the range

  -32,768 to **32,767**

Sample atoi and itoa

- For example:

```
.DATA ; Begin initialized data segment
    buffer BYTE 8 DUP (?)
…
.CODE ; Code segment
…
    input buffer, 8 ; Read zero to 6 ASCII characters
    atoi buffer ; Convert string to 2's complement number
    jno L1 ; Check the overflow flag
; Handle input error:
…
L1:
; Success: result of conversion is in AX
    itoa buffer, ax ; Convert 16-bit signed integer to string
…
```


What's Next

- Stack Operations
- Defining and Using Procedures
- Program Design Using Procedures
- Linking to an External Library
- The IO.h Library
- **64-Bit Assembly Programming**

64-Bit Assembly Programming

- Calling 64-Bit Subroutines
- The x64 Calling Convention

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.

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 IO.h library for all standard I/O and data conversion
  - Want to learn more? Study the IO.h in the class web site

55 64 67 61 6E 67 65 6E