Computer Organization and Assembly Language

Lecture 5 – Procedures

Procedures

- As programs get larger and larger, it becomes necessary to divide them into a series of ***procedures***.
- A procedure is a block of logically-related instruction that can be called by the main program or another procedure.
- Each procedure should have a single purpose and be able to do its job independent of the rest of the program.

Why Are Procedures Important?

- You need them to do input-output in assembly language.
- Procedures help you gain major insight into how the runtime stack is used.
- Your programs will grow to the point where you either divide them into procedures or you never understand the whole program.

Linking to an External Library

- Working With an external library allows you to write programs that use procedures that you do not necessarily have to know how to write.
- The Irvine32.lib is an example of such a library.

Linking and Link Libraries

- A link library is a file containing procedures that have already been assembled in machine language code. These procedures can be linked to a procedure that is written separately.
- In order to use the procedure WriteString, your program must contain

```
WriteString PROTO
```

- This informs the assembler that there is a separate procedure that will be linked to the program, which is called by writing

```
call WriteString
```

Linker Commands Options

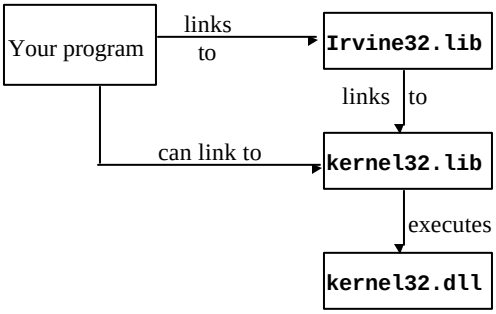
- The linker combines the programmer's object file with one or more object files and link libraries.
- To create the .exe file at the command line, type

```
link32 hello.obj irvine32.lib kernel32.lib
```

- The **`make32.bat`** file contains the command

```
link32 %1.obj irvine32.lib kernel32.lib
```

Overall Structure



The Link Library Procedures

- The Irvine32 link library contains a large collection of procedures that are useful in writing 32-bit assembly language programs.
- To understand what these procedures do, it is important to understand these terms:
  - Console - a 32-bit console windows running in Window's 32-bit text mode.
  - Standard input - the keyboard
  - Standard output - the screen

The Link Library Procedures (continued)

- **ClrScr** - Clears the screen and locates the cursor in the upper left corner

```
call    ClrScr
```

- **CrLf** - Advances the cursor to the beginning of the next line.

```
call    CrLf
```

- **Delay** - Pause the computer for x milliseconds, with x stored in the EAX register

```
mov     eax, 1000
call    Delay
```

The Link Library Procedures (continued)

- **DumpMem** - writes a range of memory to standard output in hexadecimal

```
.data
array  DWORD        1, 2, 3, 4, 5, 6, 7, 8, 9, 0ah, 0bh
.code
main   PROC
       mov    esi, OFFSET array   ; starting offset
       mov    ecx, LENGTHOF array ; # of units
       mov    ebx, TYPE array     ; double format
       call   DumpMem
```

- **DumpRegs** - Displays the contents of the EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP and EFLAGS registers in hexadecimal format.

```
call    DumpRegs
```

The Link Library Procedures (continued)

- **GetCommandTail** - copies the program command line into a null-terminated string. If the command is empty , the Carry Flag is set; if it's non-empty, the Carry Flag is cleared.
- If the command is

  Encrypt file1.txt file2.txt

  GetCommandTail would save this line, including the name of the files.

```
.data
cmdTail      BYTE   129    DUP(0)       ; empty buffer
.code
      mov    edx, OFFSET cmdTail
      call   GetCommandTail      ; filss the buffer
```

The Link Library Procedures (continued)

- **GetMSeconds**- returns the number of milliseconds that have elapsed since Midnight., placing them in the EAX register.

```
.data
StartTime   DWORD       ?
.code
      call  GetMSeconds
      mov   StartTime, eax
L1:   ; Execute a loop here
      Loop L1
      call  GetMSeconds
      sub   eax, StartTime; EAX contains loop
                          ; time in milliseconds
```

The Link Library Procedures (continued)

- **GoToXY**- moves the cursor to a given row and column. The column number (X-coordinate) is DL register and the row number (Y-Coordinate) is in the DH register.

```
mov   dh, 10      ; Row 10
mov   dl, 20      ; Column 20
call GoToXY       ; Locate cursor
call  GetMSeconds
```

The Link Library Procedures (continued)

- **Random32** – Returns a pseudorandom number which is returned in the EAX register. It uses an input called the seed, which is initialized by the **Randomize** procedure. IF you want the number within the range 0 to n-1, place n in the EAX register before calling **RandomRange**.

```
      call        Randomize
      mov         ecx, 10
L1:   call        Random32
      ; Use or Display random value in EAX
      ;     register
      mov         eax, 5000
      call        RandomRange ; Num. is 0 to 4999
      loop        L1
```

The Link Library Procedures (continued)

- **ReadChar** – Reads a single character from standard input which is returned in the AL register. the character is not echoed on the screen.

```
.data
char          BYTE   ?
.code
              call          ReadChar
              mov           char, al
```

- **ReadHex** – reads a 32-bit hexadecimal integer from standard input and returns it in the EAX register. (Either A-F or a-f can be used as digits).

```
• .data
hexval DWORD  ?
.code
              call ReadHex
              mov    HexVal, eax
```

The Link Library Procedures (continued)

- **ReadInt** – Reads a 32-bit integer from standard input which is returned in the EAX register. After the optional sign, there may only be digits.

```
• .data
intVal        DWORD         ?
.code
              call          ReadInt
              mov           intVal, eax
```

The Link Library Procedures (continued)

- **ReadString** – reads a string of characters from standard input stopping when the user presses Enter. It returns # of bytes read in the EAX register. The EDX register must contain the offset where the string is to be stored.
-

```
.data
buffer      BYTE  50 DUP (?)  ; Holds the string
byteCount   DWORD ?           ; Holds the string
                              ; length
.code
      mov   edx, OFFSET buffer ; String's Pointer
      mov   ecx, (SIZEOF buffer)-1; Max. Length
      call  ReadString        ; Read It!
      mov   byteCount, eax    ; Save # of
                              ; characters
```

The Link Library Procedures (continued)

- **SetTextColor** – sets the current foreground and background colors for text output

black= 0	red = 4	gray = 8	lightRed = 12
blue= 1	magenta = 5	lightBlue = 9	lightMagenta = 13
green = 2	brown = 6	lightGreen = 10	yellow = 14
cyan = 3	lightGray = 7	lightCyan = 11	white = 15

- The background color is multiplied by 16 and added to the foreground color.

```
mov eax, white + (blue*16); white on blue
call SetTextColor
```

The Link Library Procedures (continued)

- **WaitMsg** – displays the message "Press [Enter] to continue.." and pauses the program until the user presses the Enter key.

```
call     WaitMsg
```

- **WriteBin** – writes an integer to standard output in ASCII binary format. The value must be in the EAX register.

```
.code
     mov   eax, 12346AF9h
     call WriteBin
; displays 0001 0010 0011 0100 0110 1010 1111 1001
```

- **WriteDec** – writes a 32-bit unsigned integer to standard output that was placed in the EAX register.

```
mov    eax, 2957ffffh
call   WriteDec    ; displays "295"
```

The Link Library Procedures (continued)

- **WriteHex** – writes a 32-bit unsigned integer placed in the EAX register to standard output in 8-digit hexadecimal format. Lead zeros are inserted as necessary.

```
mov      eax, 7fffh
call     WriteHex    ; displays: "00007FFF"
```

- **WriteInt** – writes a 32-bit ***signed*** integer to standard output. The value must be in the EAX register.

```
.code
     mov   edx, 216543
     call WriteInt     ; displays: "+216543"
```

The Link Library Procedures (continued)

- **WriteString** – writes a null-terminated string to standard output. The string's offset must be placed in the EDX register.

```
.data
prompt   BYTE  "Enter your name: ", 0
.code
         mov   edx, OFFSET prompt
         call  WriteString
```

The **Irvine32.inc** Include File

```
; Include file for Irvine32.lib
  (Irvine32.inc)
INCLUDE SmallWin.inc
; MS-Windows prototypes, structures, and constants
.NOLIST

; Last update: 1/27/02
;---------------------------------------
; Procedure Prototypes
;---------------------------------------
ClrScr PROTO        ; clear the screen
Crlf PROTO          ; output carriage-return / linefeed
Delay PROTO         ; delay for n milliseconds
DumpMem PROTO       ; display memory dump
```

```
DumpRegs PROTO      ; display register dump
GetCommandTail PROTO       ; get command-line string
GetDateTime PROTO,         ; get system date and time
  startTime:PTR QWORD
GetMseconds PROTO   ; get milliseconds past midnight
Gotoxy PROTO
IsDigit PROTO ; return ZF=1 if AL is a decimal digit
Randomize PROTO     ; reseed random number generator
RandomRange PROTO   ; generate random integer in
  specified range
Random32 PROTO      ; generate 32-bit random integer
ReadInt PROTO       ; read signed integer from console
ReadChar PROTO; reach single character from console
ReadHex PROTO ; read hexadecimal integer from console
ReadString PROTO           ; read string from console
SetTextColor PROTO         ; set console text color
```

```
WaitMsg PROTO ; display wait message, wait for Enter key
WriteBin PROTO ; write integer to output in binary
                    ; format
WriteChar PROTO     ; write single character to output
WriteDec PROTO      ; write unsigned decimal integer to
                    ;output
WriteHex PROTO      ; write hexadecimal integer to
                    ; output
WriteInt PROTO      ; write signed integer to output
WriteString PROTO   ; write null-terminated string to
                    ; output

; Copy a source string to a target string.
Str_copy PROTO,
  source:PTR BYTE,
  target:PTR BYTE
```

```
; Return the length of a null-terminated string..
Str_length PROTO,
   pString:PTR BYTE

; Compare string1 to string2. Set the Zero and
; Carry flags in the same way as the CMP instruction.
Str_compare PROTO,
   string1:PTR BYTE,
   string2:PTR BYTE

; Trim a given trailing character from a string.
; The second argument is the character to trim.
Str_trim PROTO,
   pString:PTR BYTE,
   char:BYTE
```

```
; Convert a null-terminated string to upper case.
Str_ucase PROTO,
   pString:PTR BYTE

;-----------------------------------
; Standard 4-bit color definitions
;-----------------------------------
black        = 0000b
blue         = 0001b
green        = 0010b
cyan         = 0011b
red          = 0100b
magenta      = 0101b
brown        = 0110b
lightGray    = 0111b
gray         = 1000b
```

```
lightBlue    = 1001b
lightGreen   = 1010b
lightCyan    = 1011b
lightRed     = 1100b
lightMagenta = 1101b
yellow       = 1110b
white        = 1111b
.LIST
```

Library Test Program

```
TITLE Testing the Link Library      (TestLib.asm)
; Testing the Irvine32 Library
INCLUDE     Irvine32.inc
CR = 0Dh    ; Carriage Return
LF = 0Ah    ; Line Feed

.data
str1  BYTE  "Generating 20 random integers "
      BYTE  "between 0 and 990:", CR, LF, 0
str2  BYTE  "Enter a 32-bit signed integer: ", 0
str3  BYTE  "Enter your name: ", 0
str4  BYTE  "The following key was pressed: ", 0
str5  BYTE  "Displaying the registers:",
      BYTE  CR, LF, 0
str6  BYTE  "Hello, ", 0
```

```
buffer      BYTE  50 dup(?)
dwordVal    DWORD ?

.code
main  PROC
; Set text color to black text on white
  background:
  mov eax, black + (16*white)
  call      SetTextColor
  call      ClrScr            ; clear the screen
  call      Randomize   ; reset Random Number
                        ; Generator
```

```
; Generate 20 random integers between 0 and 990
; Include a 500 millisecond delay
      mov   edx, OFFSet str1 ; display message
      call  WriteString
      mov   ecx, 20     ; loop counter
      mov   dh, 2       ; screen row 2
      mov   dl, 0       ; screen column 0

L1:   call  GoToXY
      mov   eax, 991    ; indicate range+1
      call  RandomRange ; EAX = random integer
      call  WriteDec
      mov   eax, 500
      call  Delay       ; pause for 500 msec
      inc   dh          ; next screen row
      add   dl, 2       ; move 2 col.to the right
      loop  L1
```

```
      call  CrLf        ; new line
      call  WaitMsg     ; "Press [Enter]  ..."
      call  ClrScr            ; clear screen

; Input a signed decimal integer and redisplay it
; in various formats:
      mov   edx, OFFSET str2  ; "Enter a 32-..."
      call  WriteString
      call  ReadInt           ; input the integer
      mov   dwordVal, eax     ; save in memory
      call  CrLf        ; new line
      call  WriteInt    ; display as signed int.
      call  CrLf
      call  WriteHex    ; display in hexadecimal
      call  CrLf
      call  WriteBin    ; display in binary
      call  CrLf
```

```
; Display the CPU registers
      call  CrLf
      mov   edx, OFFSET str5 ; "Displaying … "
      call  WriteString
      call  DumpRegs    ; display the registers
      call  CrLf

; Display a memory dump
      mov   esi, OFFSET dwordVal    ; Start OFF.
      mov   ecx, LENGTHOF dwordVal  ; # of dwords
      mov   ebx, TYPE dwordVal  ; size of dword
      call  DumpMem           ; display memory
      call  CrLf              ; new line
      call  WaitMsg           ; "Press [Enter].."
```

```
; Ask the user to input their name:
  call      ClrScr      ; clear screen
  mov edx, OFFSET str3  ; "Enter your name": "
  call      WriteString
  mov edx, OFFSET buffer ; the buffer pointer
  mov ecx, SIZEOF buffer-1    ; max. # of chars.
  call      ReadString        ; input your name
  mov edx, OFFSET str6        ; "Hello, "
  call      WriteString
  mov edx, OFFSET buffer      ; Display your name
  call      WriteString
  call      CrLf

  exit
main  ENDP
  END main
```

Stacks

- A stack is a last-in-first-out data structure that is manipulated by means of three operations:
  - ***Push*** - add an item to the stack
  - ***Pop*** - remove the most recent item from the stack
  - ***Empty*** - true if nothing is on the stack; false if there is at least one item on the stack.

Runtime Stack

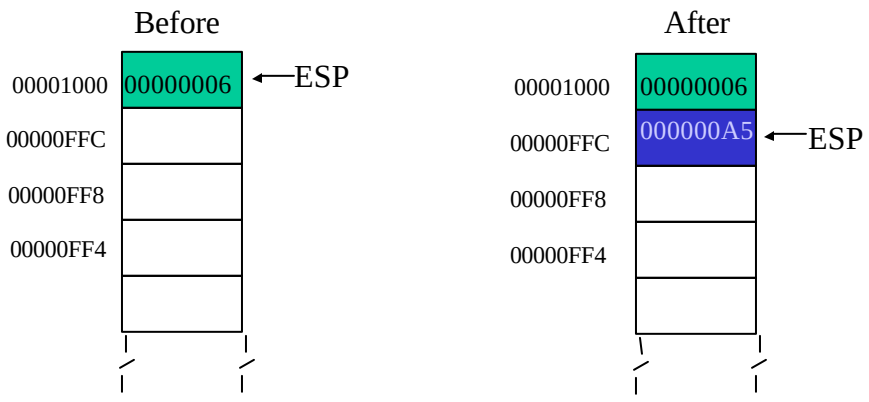
- The runtime stack is a memory array that is managed directly by the CPU using the SS and ESP registers.
- In Protected mode, the SS register holds a segment descriptor and is not modified byu user programs; the ESP register holds a 32-bit offset into some memory location on the stack.

The Intel Processor's Stack

- The stack in an Intel processor is a special memory area.
  - The stack is a temporary holding area for addresses and data.
  - Most of the data held here allows a program to return (successfully) to the calling program and procedures or to pass parameters.
  - The stack resides in the stack segment.

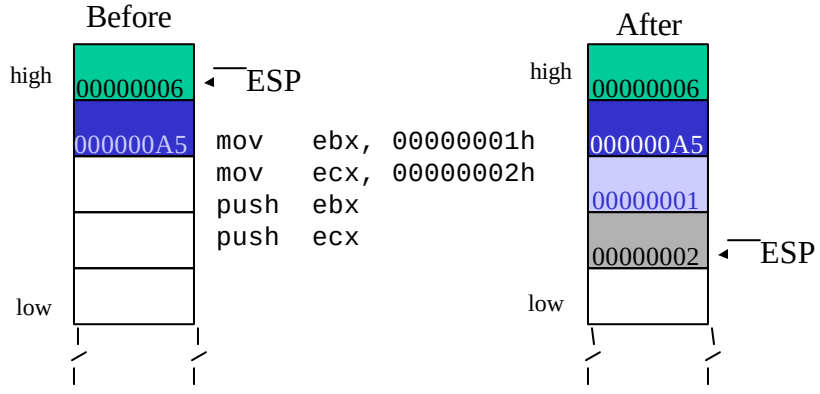
Stack Operations - Push

```
mov   eax, 00000A5h
push  eax
```

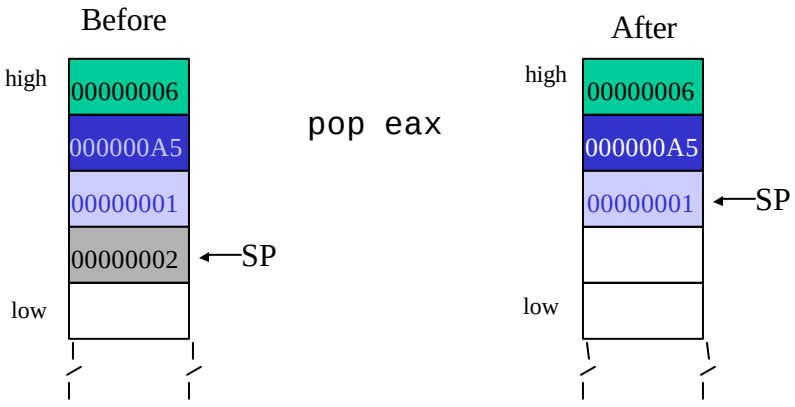


Stack Operations - Push (continued)

```
mov   ebx, 00000001h
mov   ecx, 00000002h
push  ebx
push  ecx
```



Stack Operations - Pop



Uses of the Stack

- There are several important uses of stacks in programs:
  - A stack makes an excellent temporary save area for registers, allowing a program to use them as a scratch area and then to restore them.
  - When a subroutine is called, the CPU saves a return address on the stack, allowing the program to return to the location after the procedure call.
  - When calling a procedure, you can push ***arguments*** on the stack , allowing the procedure to retrieve them.
  - High-level languages create an area on the stack inside subroutines where procedure store local variables and them discard them when it leaves the procedure.

Stack Operations - PUSH

- PUSH Instruction
  - Decrements ESP and copies a 16-bit or 32-bit register or memory operand onto the stack at the location indicated by SP.
  - With 80286+ processors, you can push an immediate operand onto the stack.
  - Examples:

```
push ax       ; push a 16-bit register operand
push ecx      ; push a 32-bit register operand
push memval   ; push a 16-bit memory operand
push 1000h    ; push an immediate operand
```

Stack Operations - POP

- POP Instruction
  - copies the contents of the stack pointed to by SP into a register or variable and increments SP.
  - CS and IP cannot be used as operands.
  - Examples:

```
pop cx      ; pop stack into 16-bit register
pop memval; pop stack into 16-bit memory
  operand
pop eds     ; pop stack into 32-bit register
```

Other Stack Operations – PUSHFD & POPFD

- PUSHFD and POPFD Instructions
  - PUSHFD pushes the EFLAGS register onto the stack, preserving it in case it changes.
  - POPFD restores the EFLAGS registers.
  - Example

```
pushfd                ; save the flags
call display_sub      ;call a subroutine
popfd                 ; restore the flags
```

Other Stack Operations – PUSHA & PUSHAD

- ***PUSHA*** (286+) pushes AX, CX, DX, BX, SP, BP, SI and DI onto the stack in the above order.
- ***POPA*** restores the registers saved using PUSHA
- ***PUSHAD*** (386+) pushes EAX, ECX, EDX, EBX, ESP, EBP, ESI and EDI onto the stack in the above order.
- ***POPAD*** restores the registers saved using PUSHAD.

Example: Reversing A String

```
TITLE Reversing a String      (RevStr.asm)
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
      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
      mov   aName[esi], al
      inc   esi
      loop  L2

; Display the name
      mov   edx, OFFSET aName
      call  WriteString
      call  CrLf

      exit
main  ENDP
      END   main
```

Procedures

- In general, there are two types of subprograms: functions and procedures (or subroutines).
  - ***Functions*** return a value (or ***result***).
  - ***Procedures*** (or ***subroutines***) do not.
  - The terms procedures and subroutines are used interchangeably although some languages use one term and others use the other.
  - Calling a procedure implies that there is a return. Also implies is that the state of the program, (register values, etc.) are left unaffected when the program returns to the calling procedure or program.

PROC and ENDP Directives

- PROC and ENDP mark the beginning and end of a procedure respectively.

```
.code
main proc
  … …
  call   MySub
  … …
  main endp
                ;  Nb: procedures cannot overlap
  MySub  proc   ; one must have endp before
  …             ; the next can have proc
  ret
  MySub endp
```

Nested procedure calls

- A procedure may call other procedures.
- The list of return addresses (as well as other data) is saved on the stack, with the most recently called procedure's return address and data on top.
-

```
main    proc            sub2    proc
        call sub1       …
        mov  eax, …     call    sub3
        …                       ret
        main endp               sub2 endp
        …
sub1    proc            sub3    proc
        call sub2               …
        ret                     ret
        sub1 endp               sub3  endp
```

The **`exit`** Instruction

- While all other procedures end with the **`ret`** instruction, **`exit`** is used by the main procedure.
- **`exit`** is actually an not an instruction but an alias for

  `INVOKE ExitProcess, 0`

  the Windows system function for terminating programs
- In Irvine16.inc, it is defined as

```
mov     ah, 4ch
int     21h
```

Local and Global Labels

- Be default, code labels have ***local*** scope, limited to the procedures in which they are located.
- By ending a label with a double colon, the scope become ***global*** and it can be referenced outside the current procedure.

```
main    PROC
        jmp   L2    ; error!
L1::    exit        ; global label
        main  endp
sub2    PROC
L2:                 ; local label
        jmp   L1    ; OK
        ret
sub2          endp
```

Passing Parameters

- Passing arguments in registers
  - The most common method for passing parameter between the calling program (or procedure) and the procedures that it calls is through the registers
  - It is efficient because the called procedure has immediate and direct use of the parameters and registers are faster than memory.
  - Example: WriteInt

```
.data
aNumber     DWORD       234
.code
            mov         eax, aNumber
            call        WriteInt
```

Preserving Registers

- Ordinarily procedures have the responsibility to preserve register contents.
  - This ensures that the main procedure has no surprises.
  - What would happen here if WriteInt modified CX?

```
.data
DECIMAL_RADIX = 10
LIST_COUNT=20
aList         dw     LIST_COUNT dup(?)
.code
    mov       ecx, LIST_COUNT
    mov       ebx, DECIMAL_RADIX
    mov       esi, offset aList
L1: mov       eax, [si]
    call      WriteInt
    add       esi, size aList
    loop      L1
```

Using Registers to Return a Value

- Some functions will use a register as a method of returning a value to the calling procedure:

```
SumOf     proc
          push eax
          mov  eax, ebx
          add  eax, ecx
          pop  eax  ; Error – AX reset to orig. value
          ret
SumOf     endp
```

Procedure **ArraySum**

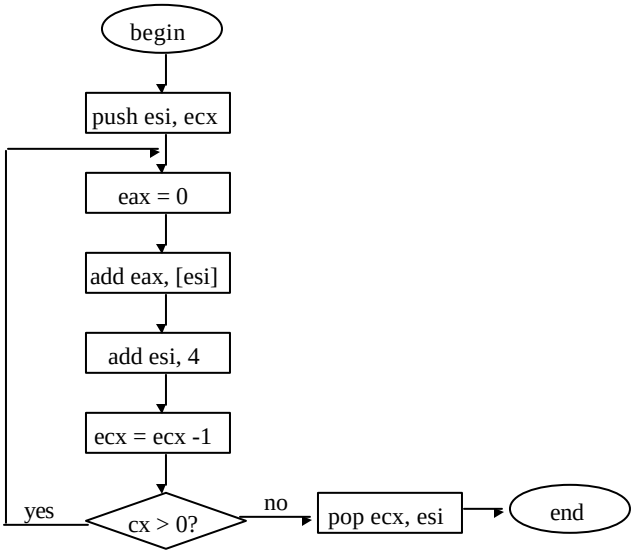
```
ArraySum PROC
;-----------------------------------------
;  Calculates the sum of an array of 32-bit integers.
;  Receives:  ESI - the array offset
;             ECX = # of elements in array
;  Returns           EAX - the sum of the array
;-----------------------------------------
   pushesi           ; save ESI, ECX
   pushecx
   mov eax, 0        ; Sum = 0
L1:    add    eax, [esi]    ; Sum = Sum + x[i]
   add esi, 4        ; Point to next integer
   loopL1            ; Repeat for array size

   pop ecx
   pop esi
   ret
ArraySum ENDP
```

Calling **ArraySum**

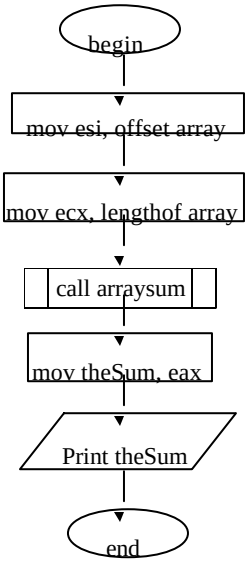
```
TITLE Driver for Array Sum    (ArrayDr.asm)
INCLUDE     Irvine32.inc
.data
array       DWORD 10000h, 20000h, 30000h, 40000h
theSum      DWORD ?
.code
main  PROC
   mov      esi, OFFSET array ; ESI points to array
   mov      ecx, LENGTHOF array     ; ECX = array
   count
   call     ArraySum          ; calculate the sum
   mov      theSum, eax       ; returned in EAX
   call     WriteHex          ; Is it correct?
   exit
main  ENDP
ArraySum PROC… …
   END main
```

Procedure goes here

Flowchart For **ArraySum**



Flowchart For **ArrayDr**



USES Operator

```
ArraySum PROC USES esi  ecx
; ESI, ECX automatically saved
      mov   eax, 0            ; Sum = 0
L1:   add   eax, [esi]  ; Sum = Sum + x[i]
      add   esi, 4      ; Point to next integer
      loop  L1          ; Repeat for array size

; ECX, ESI automatically popped
      ret
ArraySum ENDP
```

Example: Returning A Value

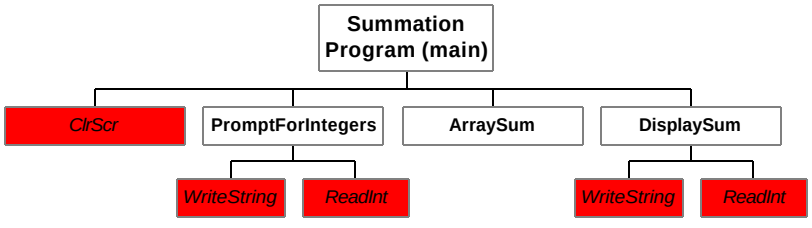
```
SumOf       PROC              ; sum of 3 integers
            push  eax         ; Save EAX
            add   eax, ebx    ; Calculate the sum
            add   eax, ecx    ; of EAX, EBX ECX
            pop   eax         ; Lost the Sum!!!
            ret
SumOf       ENDP
;  We DON'T pop the register with the return
;  value
```

Integer Summation Program Pseudocode

- In designing larger programs, dividing the specific tasks into separate procedures in extremely helpful:

```
Main
    ClrScr              ; Clear Screen
    PromptForIntegers
          WriteString   ; Display Prompt Message
          ReadInt       ; Input Integer
    ArraySum            ; Sum the Integer
    DisplaySum
          WriteString   ; Display output message
          WriteInt      ; Display Integers
```

Structure Chart



Procedure Stub

```
;-----------------------------------------------
ArraySum     PROC
;
; Calculates the sum of an array of 32-bit integers
; Receives: ESI points to the array, ECX = array size
; Returns:  EAX = sum of the array elements
;-----------------------------------------------

   ret               ; Sum is in EAX

ArraySum     ENDP
```

Sum2 Program

```
TITLE Integer Summation Program (Sum2.asm)
; This program inputs multiple integers from the
  user,
; stores them in an array, calculates the sum of the
; array and displays the sum

INCLUDE     Irvine32.inc

IntegerCount = 3  ; array size

.data
prompt1     BYTE  "Enter a signed integer: ", 0
prompt2     BYTE  "The sum of the integers is : ", 0
array       DWORD IntegerCount DUP(?)
```

```
.code
main  PROC
  call      ClrScr
  mov esi, OFFSET array
  mov ecx, IntegerCount
  call      PromptForIntegers
  call      ArraySum
  call      DisplaySum
  exit
main  ENDP
```

```
;-----------------------------------------------
PromptForIntegers PROC
;
; Prompts the user for an array of integers and fills
; the array wioth the user's input.
; Receives: ESI points to the array, ECX = array size
; Returns: nothing
;-----------------------------------------------
  pushad                        ; save all registers

      mov    edx, OFFSET prompt1 ; prompt address
L1:   call   WriteString         ; display prompt
      call   ReadInt             ; Read next integer
      call   CrLf                ; go to next line
      mov    [esi], eax          ; store in array
      add    esi, 4              ; point to next int.
      loop   L1                  ; repeat
      popad                      ; restore registers
      ret
PromptForIntegers ENDP
```

```
;-----------------------------------------------
ArraySum     PROC
;
; Calculates the sum of an array of 32-bit integers
; Receives: ESI points to the array, ECX = array size
; Returns:  EAX = sum of the array elements
;-----------------------------------------------
       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, 4        ; point to next integer
       loop   L1            ; repeat for array size

       pop    ecx           ; Restore ECX, ESI
       pop    esi
       ret                  ; Sum is in EAX

ArraySum     ENDP
```

```
;-----------------------------------------------
DisplaySum   PROC
;
; Displays the sum on the screen
; Receives: EAX  = the sum
; Returns:  Nothing
;-----------------------------------------------

       push   edx                 ; Save EDX
       mov    edx, OFFSET Prompt2 ; Display message
       call   WriteString
       call   WriteInt            ; Display EAX
       call   CrLf

       pop    edx                 ; Restore EDX
       ret
DisplaySum   ENDP
  END main
```