Assembly Programming

Chapter 5: Procedures

CSE3030

Prof. Youngjae Kim

Distributed Computing and Operating Systems Laboratory (DISCOS)

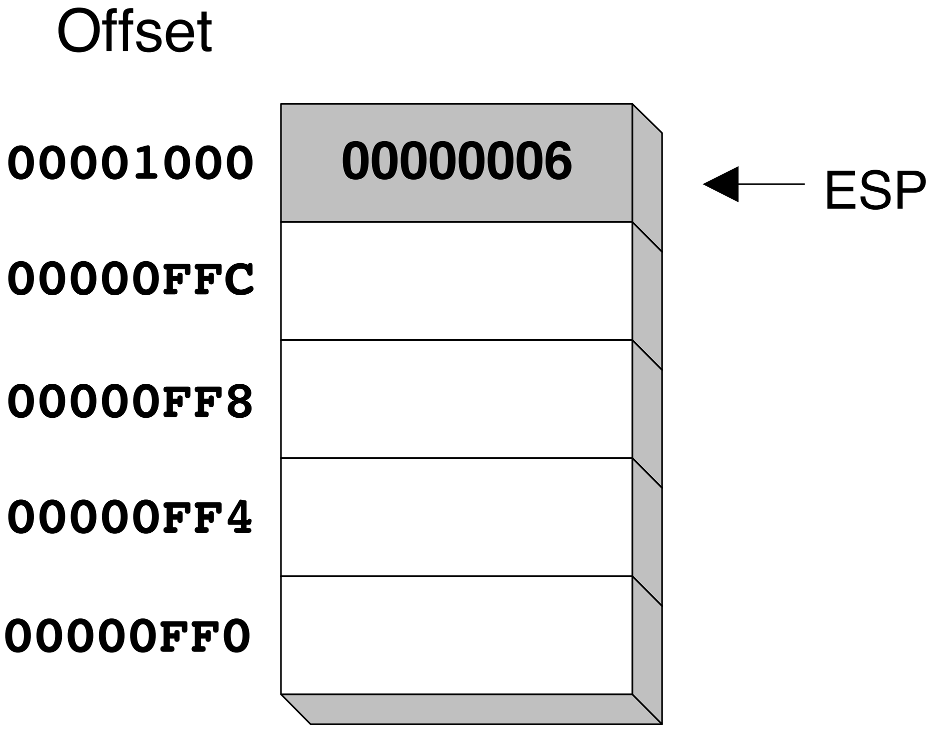
https://discos.sogang.ac.kr

Chapter 5: Procedures

- Stack Operations
  - Runtime Stack (32-Bit Mode)
  - PUSH and POP Instructions
- Defining and Using Procedures
  - PROC Directive, CALL and RETURN Instructions
  - Nested Procedure Calls
  - Passing Register Arguments to Procedures
  - Saving and Restoring Registers
- Linking to an External Library
  - Background Information
- The Irvine32 Library
  - Motivation for Creating the Library
  - Individual Procedure Descriptions

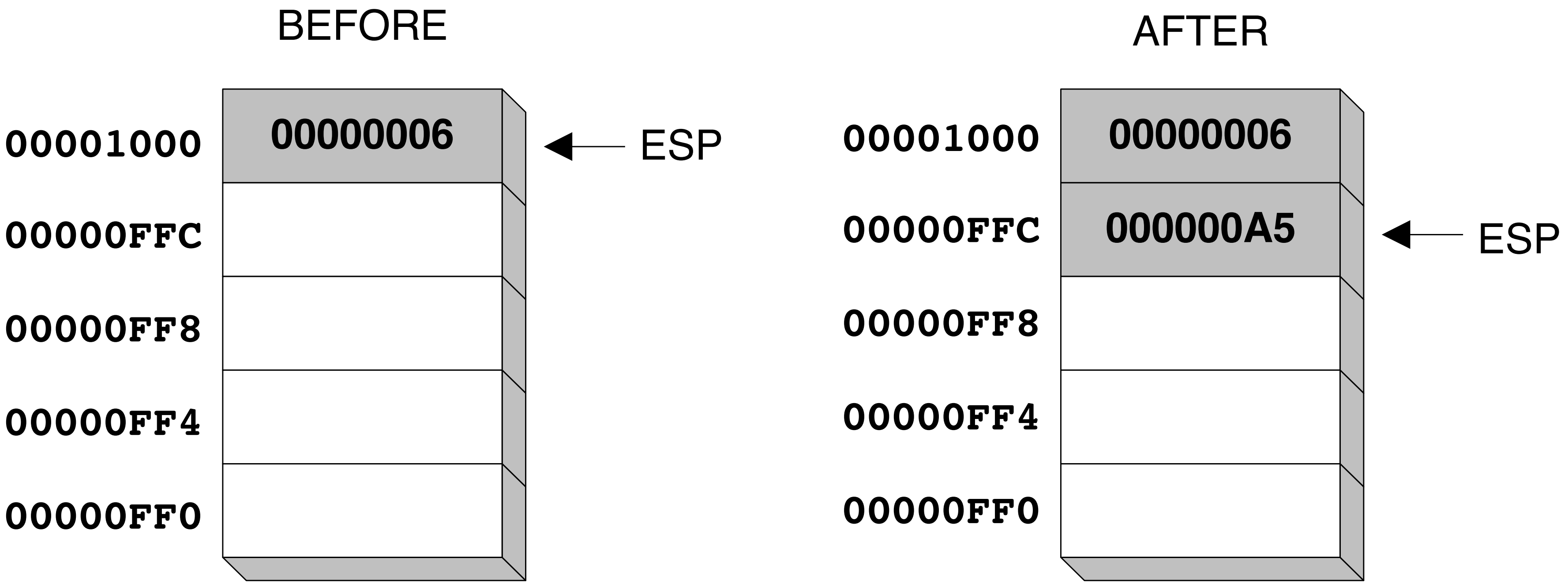
5.1 Runtime Stack

- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) (SP in Real-address mode)



PUSH Operation

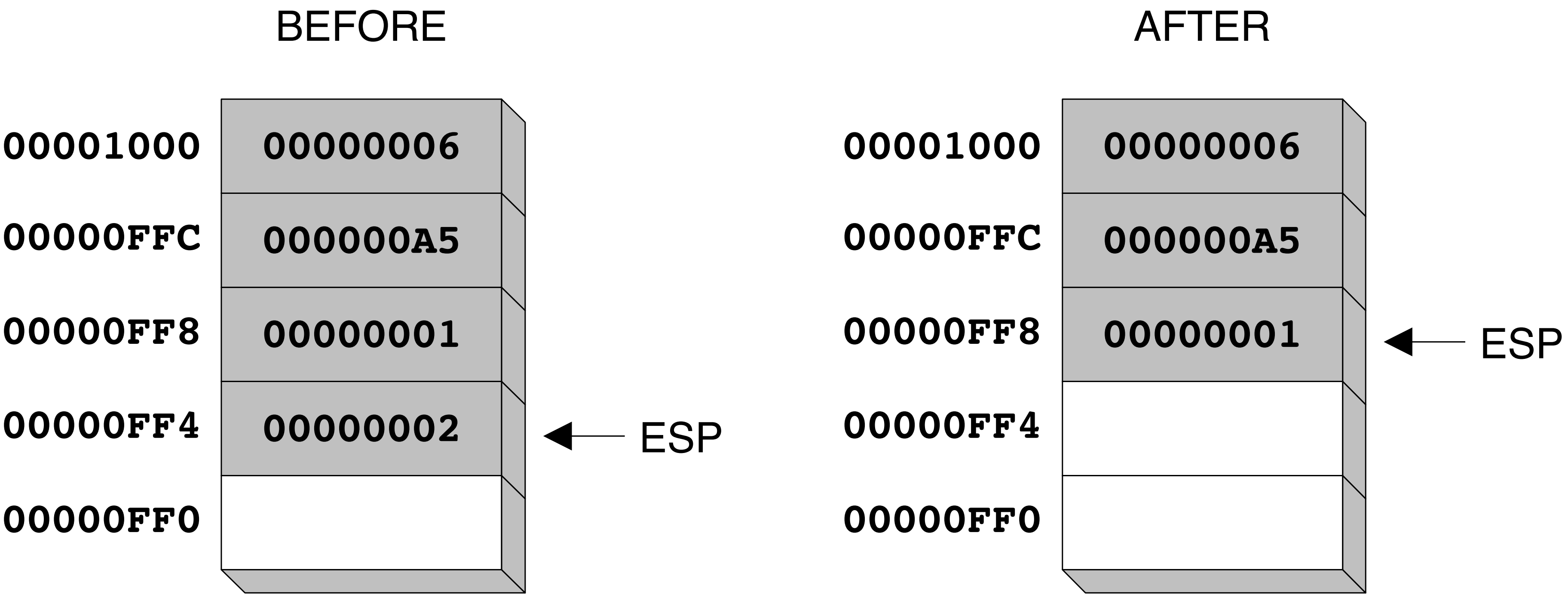
- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.



- 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.
- Removes a value from the stack.
    - After the value is popped from the stack, the stack pointer is incremented by the stack element size to point to the next-highest location in the stack.



BEFORE

00001000	00000006	
00000FFC	000000A5	
00000FF8	00000001	
00000FF4	00000002	← ESP
00000FF0		

AFTER

00001000	00000006	
00000FFC	000000A5	
00000FF8	00000001	← ESP
00000FF4		
00000FF0		

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
```

Nested 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
```

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

Reversing a String (continued)

```
.data
aName BYTE “I like StarII",0
nameSize = ($ - aName) - 1      ; nameSize = 13

.code

      mov ecx,nameSize
      mov esi,0
L1:   movzx eax,aName[esi] ; get character
      push eax             ; push on stack
      inc esi
      Loop L1

      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

5.2 Defining and Using Procedures

- 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

- A description of all tasks accomplished by the procedure.
  - Receives: Input parameters, their usage and requirements.
  - Returns: Values returned by the procedure.
  - Requires: Optional list of requirements that must be satisfied before the procedure is called.

```
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

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

00000040 is the offset of the first instruction inside MySub

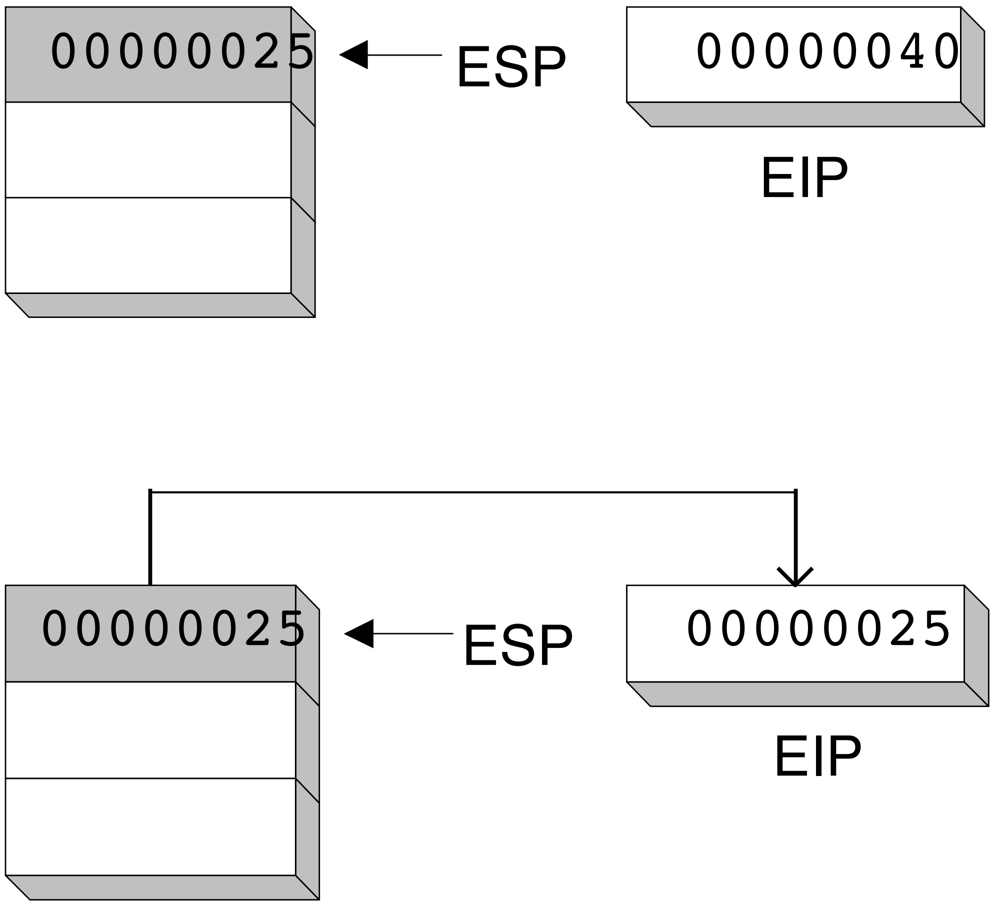
```
main PROC
   00000020 call MySub
   00000025 mov eax,ebx
   .
main ENDP

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

More Illustrations

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

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



(stack shown before RET executes)

Nested Procedure Calls

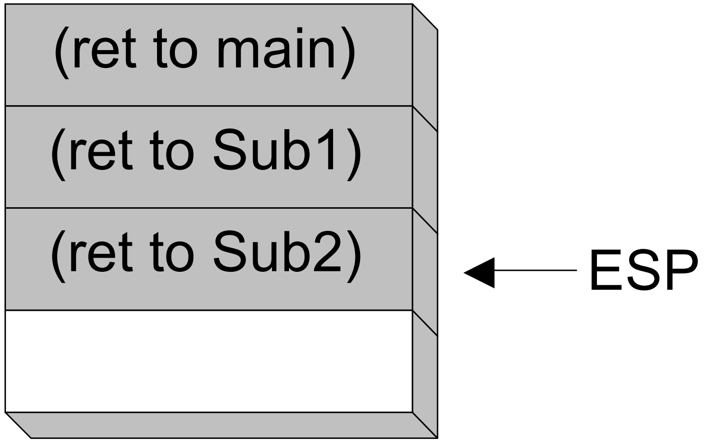
```
main PROC
     .
     .
     call Sub1
     exit
main ENDP

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
     exit
main ENDP

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

Procedure Parameters

- 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

A Procedure which is not flexible

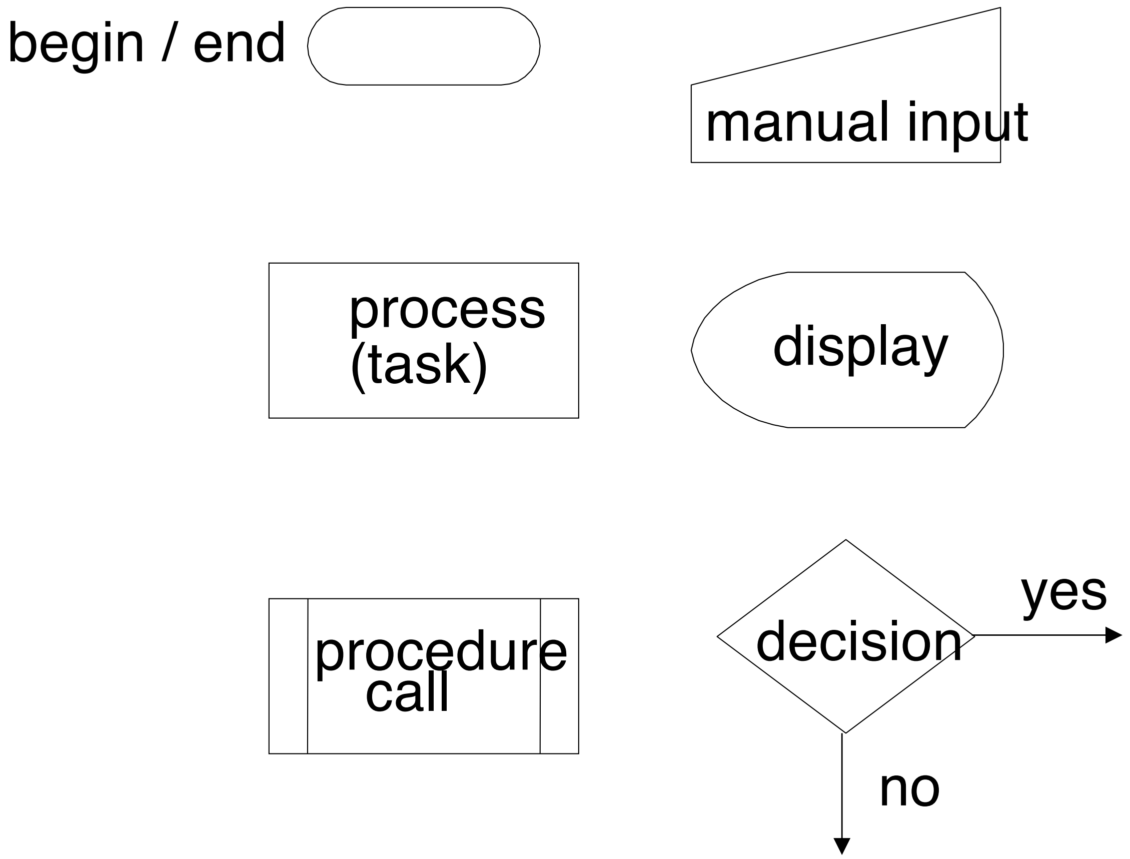
```
ArraySum PROC
    mov esi,0                ; array index
    mov eax,0                ; set the sum to zero
L1: add eax,myArray[esi]     ; add each integer to sum
    add esi,4                ; point to next integer
    loop L1                  ; repeat for array size
    mov theSum,eax           ; store the sum
    ret
ArraySum ENDP
```

Better Version

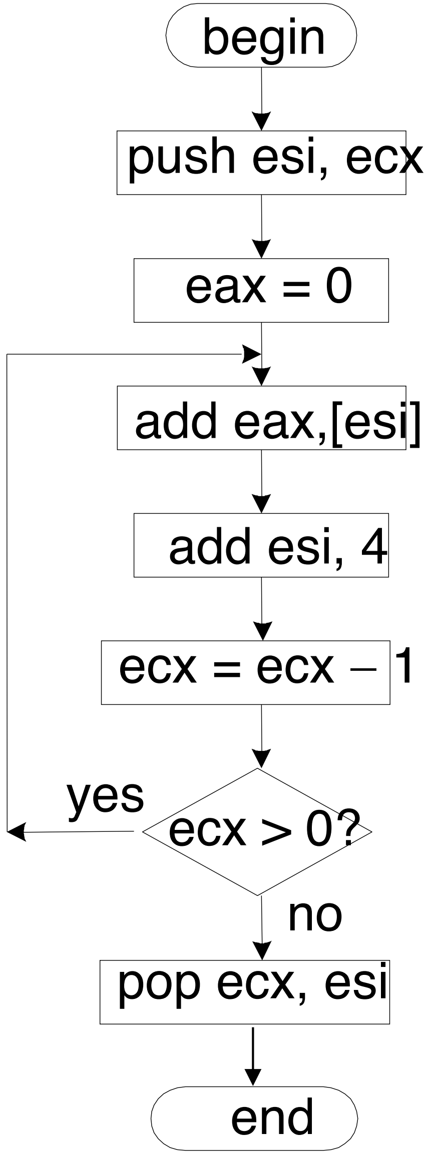
```
ArraySum PROC
; Receives: ESI points to an array of doublewords,
;           ECX = number of array elements.
; Returns: EAX = sum
    mov eax,0               ; set the sum to zero
L1: add eax,[esi]           ; add each integer to sum
    add esi,4               ; point to next integer
    loop L1                 ; repeat for array size
    ret
ArraySum ENDP
```

Program Design Using Flowchart

- Basic building blocks of flowcharts



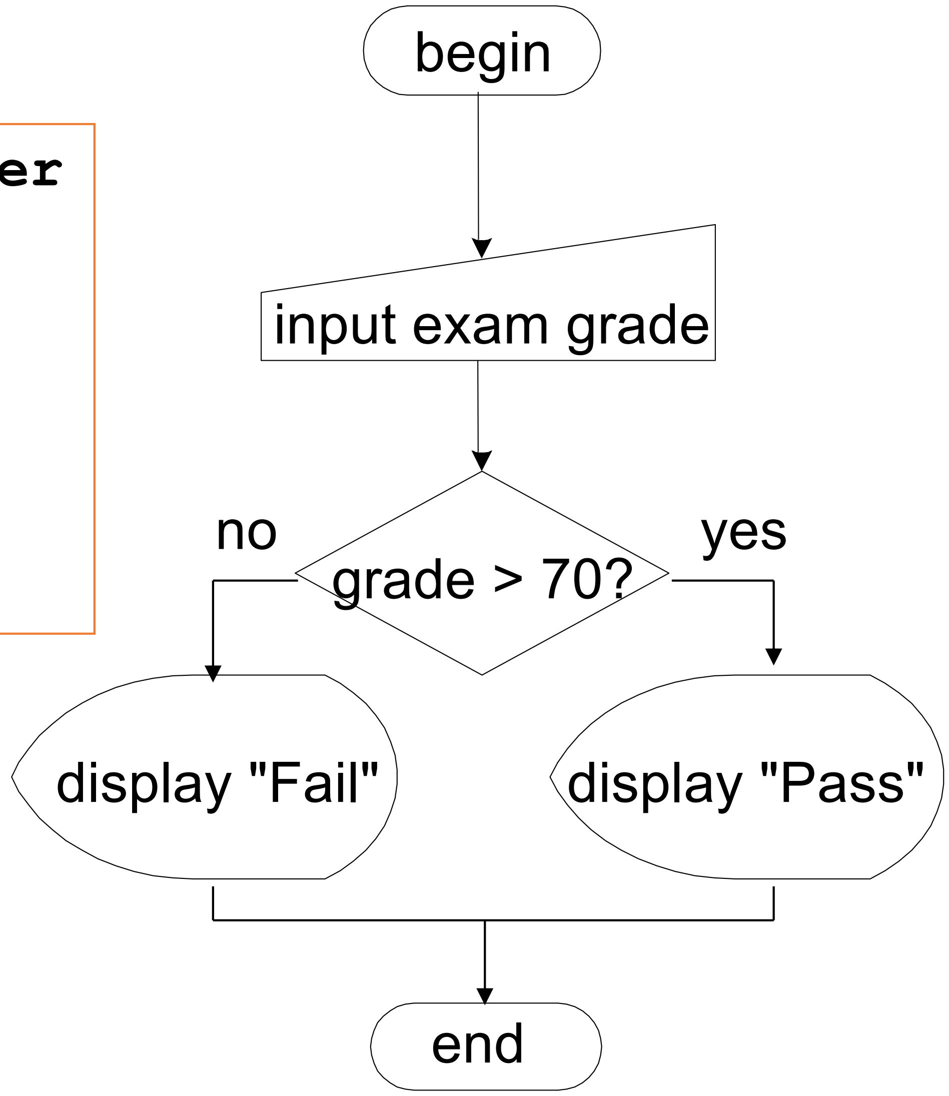
Flowchart for ArraySum procedure



```
      push esi
      push ecx
      mov  eax,0
AS1:  add  eax,[esi]
      add  esi,4
      loop AS1
      pop  ecx
      pop  esi
```

Another Flowchart Example

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



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
   mov  eax, 0
   . . .
   pop ecx
   pop esi
   ret
ArraySum ENDP
```

When not to push a register

- The sum of the three registers is stored to EAX at line (3), but the POP instruction replaces it with the starting value of EAX at 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
```

Program Design Using Procedures

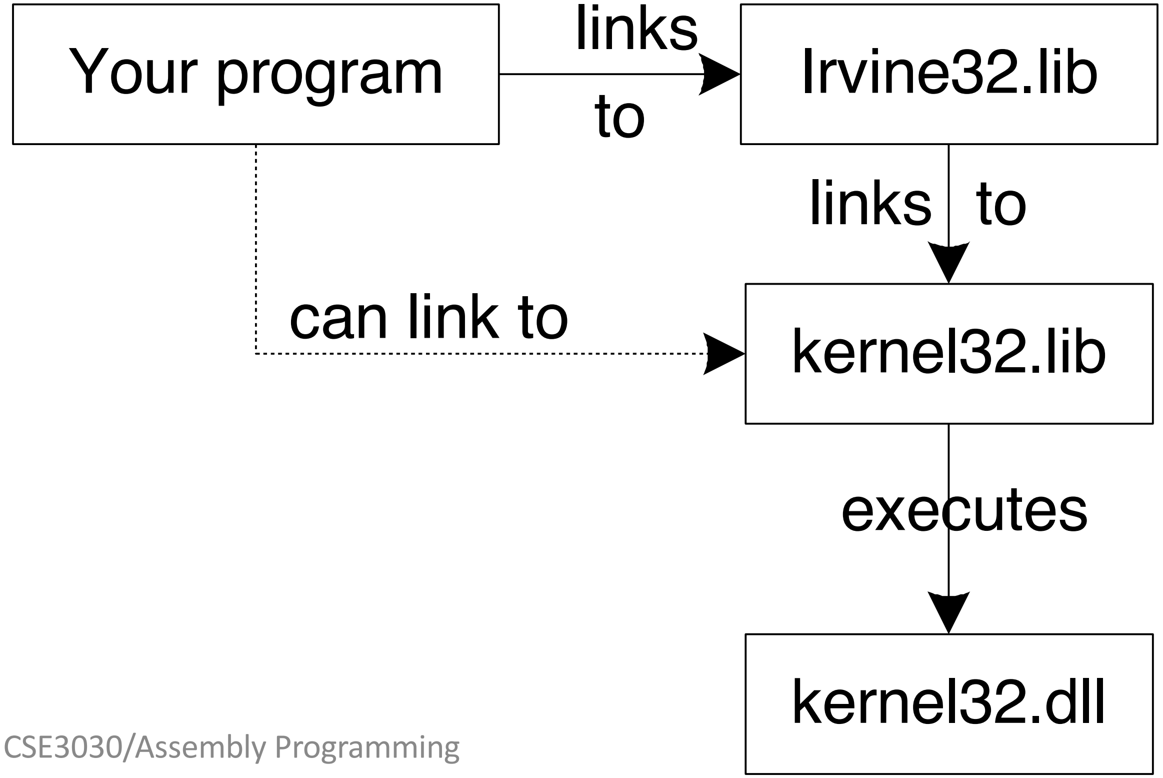
- Top-Down Design (functional decomposition) involves the following:
  - Design your program before starting to code.
  - Break large tasks into smaller ones.
  - Use a hierarchical structure based on procedure calls.
  - Test individual procedures separately.

5.3 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
  - See help file by typing “LIB /HELP” in DOS mode.
  - Or, may search msdn library.

5.3 Link 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)*

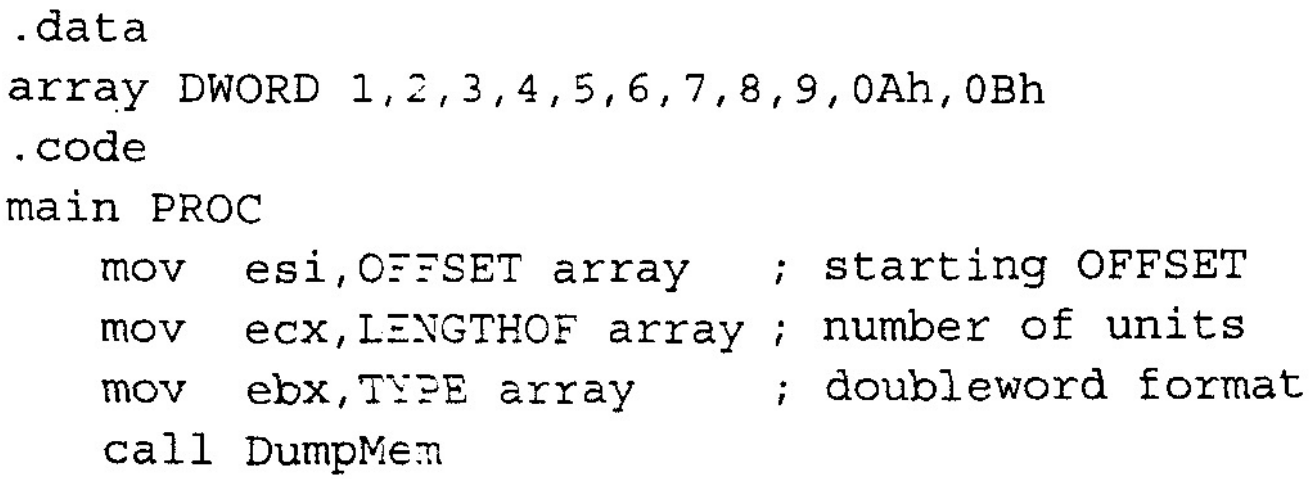


The Book's Link Library

- **`Clrscr`** - Clears the console and locates the cursor at the upper left corner.
- **`Crlf`** - Writes an end of line sequence to standard output.
- **`Delay`** - Pauses the program execution for a specified *n* millisecond interval.

```
mov    eax, 1000 ; 1 sec
call   Delay
```

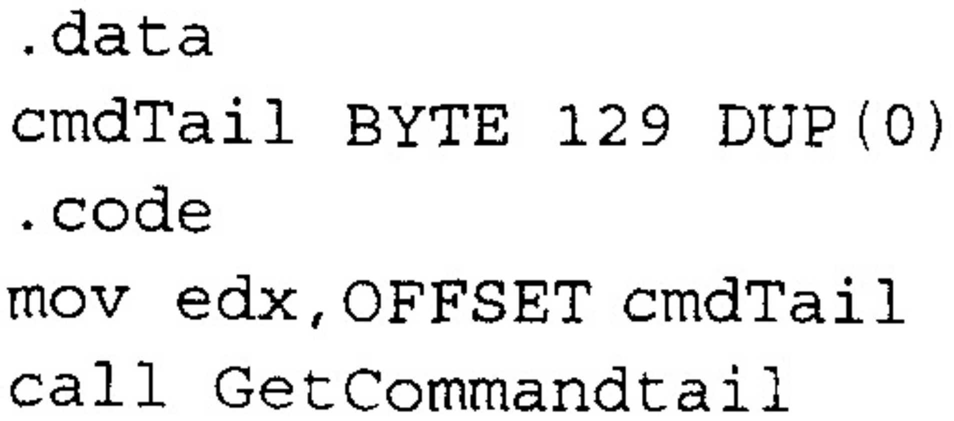
- **`DumpMem`** - Writes a block of memory to standard output in hexadecimal.



```
.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 ; number of units
    mov   ebx,TYPE array      ; doubleword format
    call  DumpMem
```

The Book’s Link Library

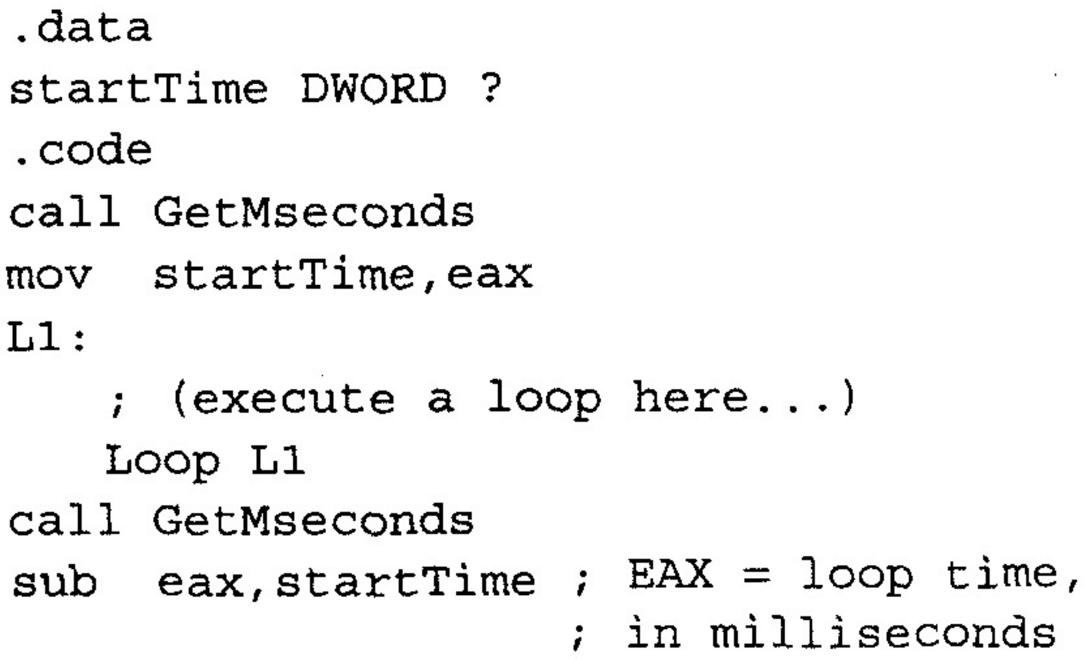
- **DumpRegs** - Displays the EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, and EIP registers in hexadecimal. Also displays the Carry, Sign, Zero, and Overflow flags.
- **GetCommandtail** - Copies the program’s command-line arguments (called the *command tail*) into an array of bytes. If empty, the Carry flag is set. Otherwise, reset.



```
.data
cmdTail BYTE 129 DUP(0)
.code
mov edx,OFFSET cmdTail
call GetCommandtail
```

The Book's Link Library

- **GetMseconds** - Returns the number of milliseconds that have elapsed since midnight.



```
.data
startTime DWORD ?
.code
call GetMseconds
mov  startTime,eax
L1:
    ; (execute a loop here...)
    Loop L1
call GetMseconds
sub  eax,startTime ; EAX = loop time,
                   ; in milliseconds
```

- **Gotoxy** - Locates cursor at row and column on the console.

The Book's Link Library

- **`Random32`** - Generates a 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.
- **`ReadHex`** - Reads a 32-bit hexadecimal integer from standard input, terminated by the Enter key.
- **`ReadInt`** - Reads a 32-bit signed decimal integer from standard input, terminated by the Enter key.
- **`ReadString`** - Reads a string from standard input, terminated by the Enter key.

The Book’s Link Library

- **`SetTextColor`** - Sets the foreground and background colors of all subsequent text output to the console.
- **`WaitMsg`** - Displays message, waits for Enter key to be pressed.
- **`WriteBin`** - Writes an unsigned 32-bit integer to standard output in ASCII binary format.
- **`WriteChar`** - Writes a single character to standard output.
- **`WriteDec`** - Writes an unsigned 32-bit integer to standard output in decimal format.
- **`WriteHex`** - Writes an unsigned 32-bit integer to standard output in hexadecimal format.
- **`WriteInt`** - Writes a signed 32-bit integer to standard output in decimal format.
- **`WriteString`** - Writes a null-terminated string to standard output.

Examples

- Example 1

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

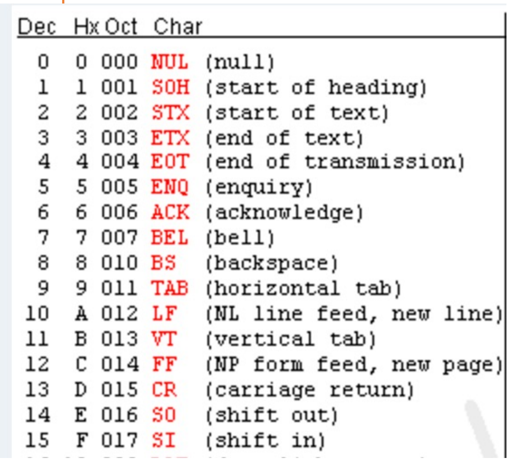
Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags.

- Example 2

```
.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.

ASCII Table



Dec	Hx	Oct	Char	
0	0	000	NUL	(null)
1	1	001	SOH	(start of heading)
2	2	002	STX	(start of text)
3	3	003	ETX	(end of text)
4	4	004	EOT	(end of transmission)
5	5	005	ENQ	(enquiry)
6	6	006	ACK	(acknowledge)
7	7	007	BEL	(bell)
8	8	010	BS	(backspace)
9	9	011	TAB	(horizontal tab)
10	A	012	LF	(NL line feed, new line)
11	B	013	VT	(vertical tab)
12	C	014	FF	(NP form feed, new page)
13	D	015	CR	(carriage return)
14	E	016	SO	(shift out)
15	F	017	SI	(shift in)

- Example 2 : Another implementation

0Dh: Carriage return

0Ah: Line feed

```
.data
str1 BYTE "Assembly language is easy!",0Dh,0Ah,0
.code
   mov  edx,OFFSET str1
   call WriteString
```

- 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 ; 0 is appended
   call ReadString
```

- 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.
  - 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
```

An Example (Integer Summation)

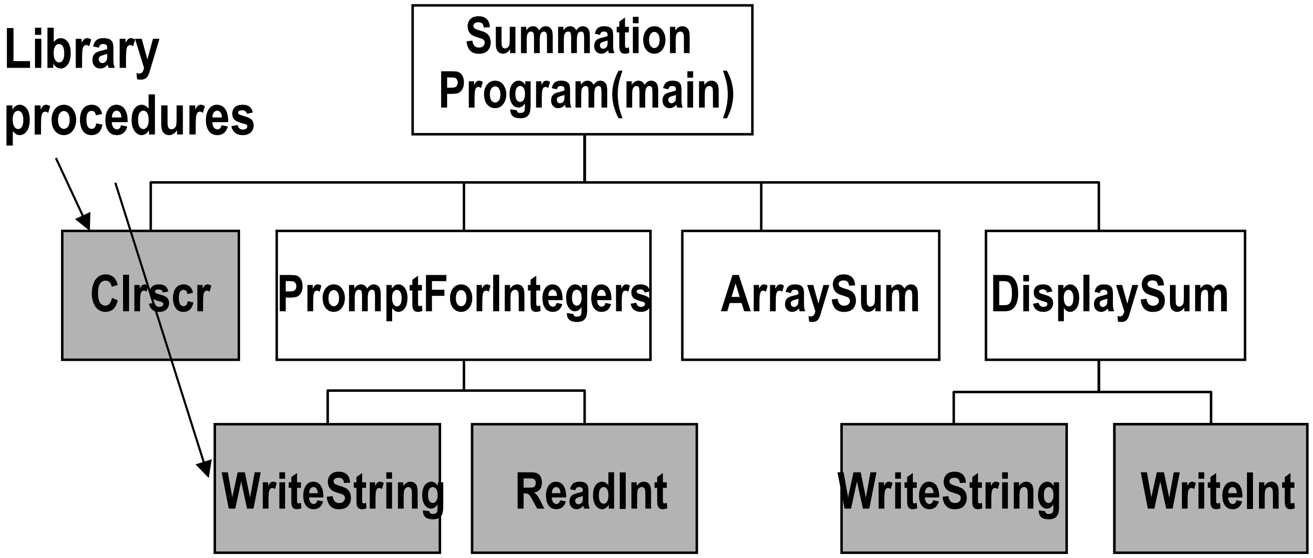
Main steps:

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

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.

Procedure Design

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



Input and Output Format Display

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

Stub Program

```
INCLUDE Irvine32.inc

.data
first DWORD 2323423424
second BYTE "adjaslfdjsl"

.code
main PROC
; Main program control procedure.
; Calls: Clrscr, PromptForIntegers,
;        ArraySum, DisplaySum

 exit
main ENDP
```

```
;------------------------------------------
PromptForIntegers PROC
;
; Prompts the user for an array of integers,
; and fills the array with the user's input.
; Receives: ESI points to an array of
;   doubleword integers, ECX = array size.
; Returns: the array contains the values
;   entered by the user
; Calls: ReadInt, WriteString
;------------------------------------------
  ret
PromptForIntegers ENDP

;------------------------------------------
ArraySum PROC
;
; Calculate the sum of an array of 32-bit ints.
; Receives: ESI points to the array,
;           ECX = array size
; Returns:  EAX = sum of the array elements
;------------------------------------------
  ret
ArraySum ENDP
```

```
;----------------------------------------------
DisplaySum PROC
;
; Displays the sum on the screen
; Recevies: EAX = the sum
; Calls: WriteString, WriteInt
;----------------------------------------------
      ret
DisplaySum ENDP

END main
```

A Complete Program(1/4)

```
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
```

A Complete Program(2/4)

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

      mov  edx,OFFSET prompt1 ; address of the prompt
L1:
      call WriteString        ; display string
      call ReadInt            ; read integer into EAX
      call Crlf               ; go to next output line
      mov  [esi],eax          ; store in array
      add  esi,4              ; next integer
      loop L1
L2:
      popad       ; restore all registers
      ret
PromptForIntegers ENDP
```

A Complete Program(3/4)

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

L2:
      pop   ecx         ; restore ECX, ESI
      pop   esi
      ret         ; sum is in EAX
ArraySum ENDP
```

A Complete Program(4/4)

```
;-----------------------------------------------------
DisplaySum PROC
;
; Displays the sum on the screen
; Recevies: EAX = the sum
; Returns:  nothing
;-----------------------------------------------------
      push edx
      mov  edx,OFFSET prompt2  ; display message
      call WriteString
      call WriteInt            ; display EAX
      call Crlf
      pop  edx
      ret
DisplaySum ENDP
END main
```