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 having been satisfied, the procedure's creator makes no promise that it will work.

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 [1/2]

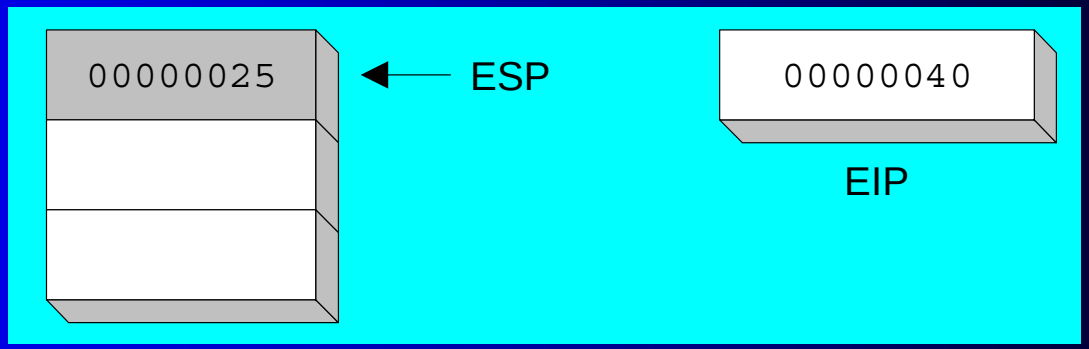
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

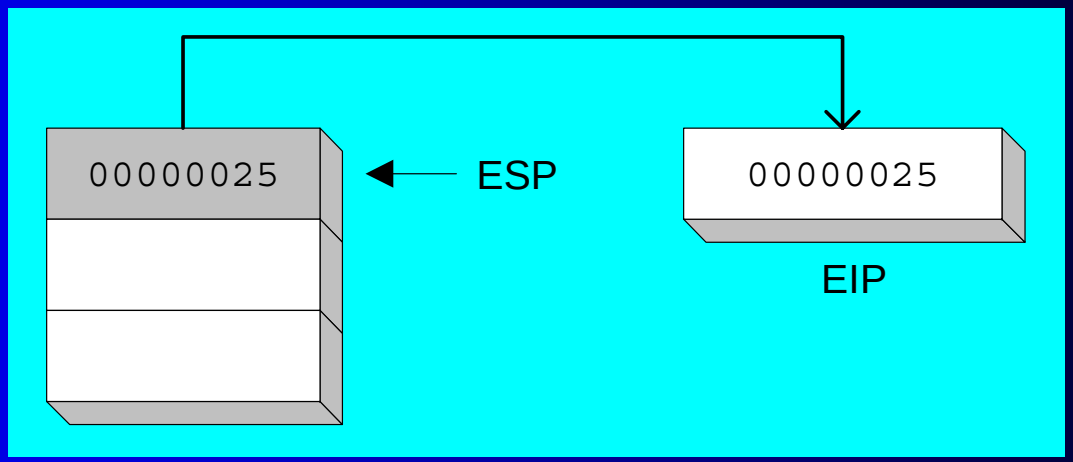
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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

CALL-RET Example [2/2]

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



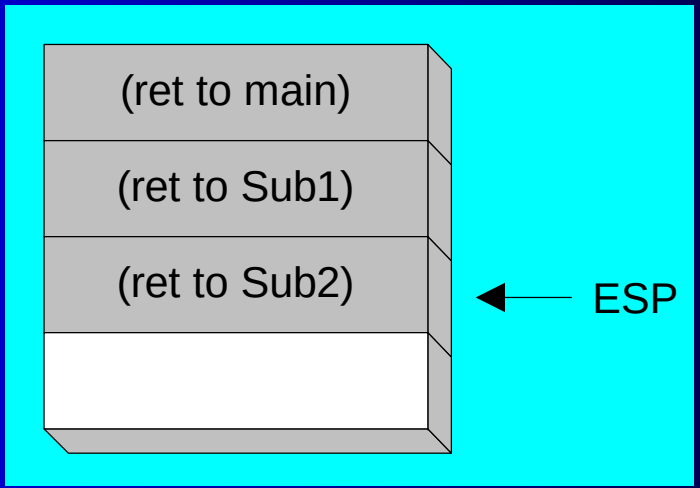
The RET instruction pops 00000025 from the stack into EIP



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 [1/3]

- 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 [2/3]

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

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
```

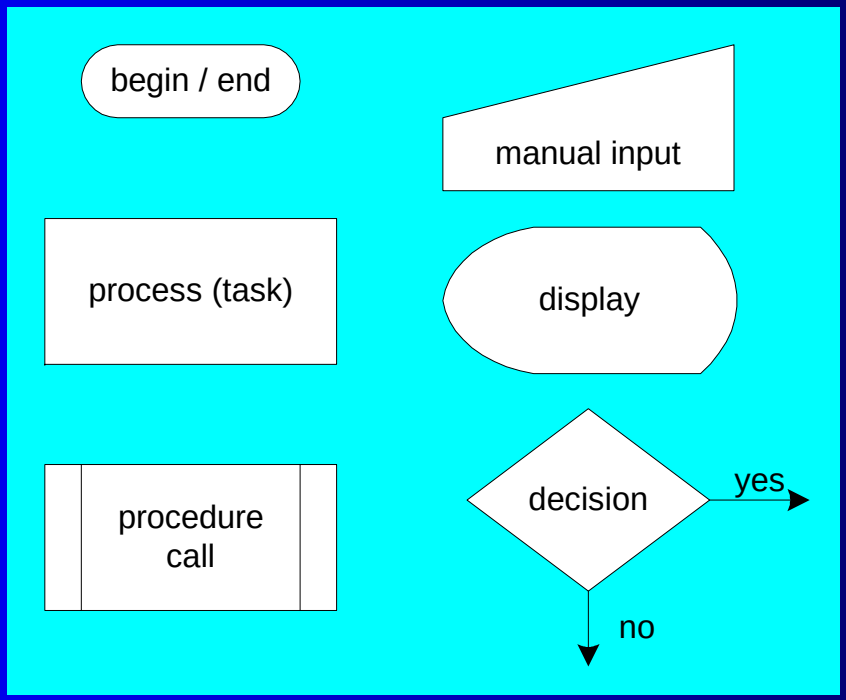
Procedure Parameters [3/3]

This version of ArraySum returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

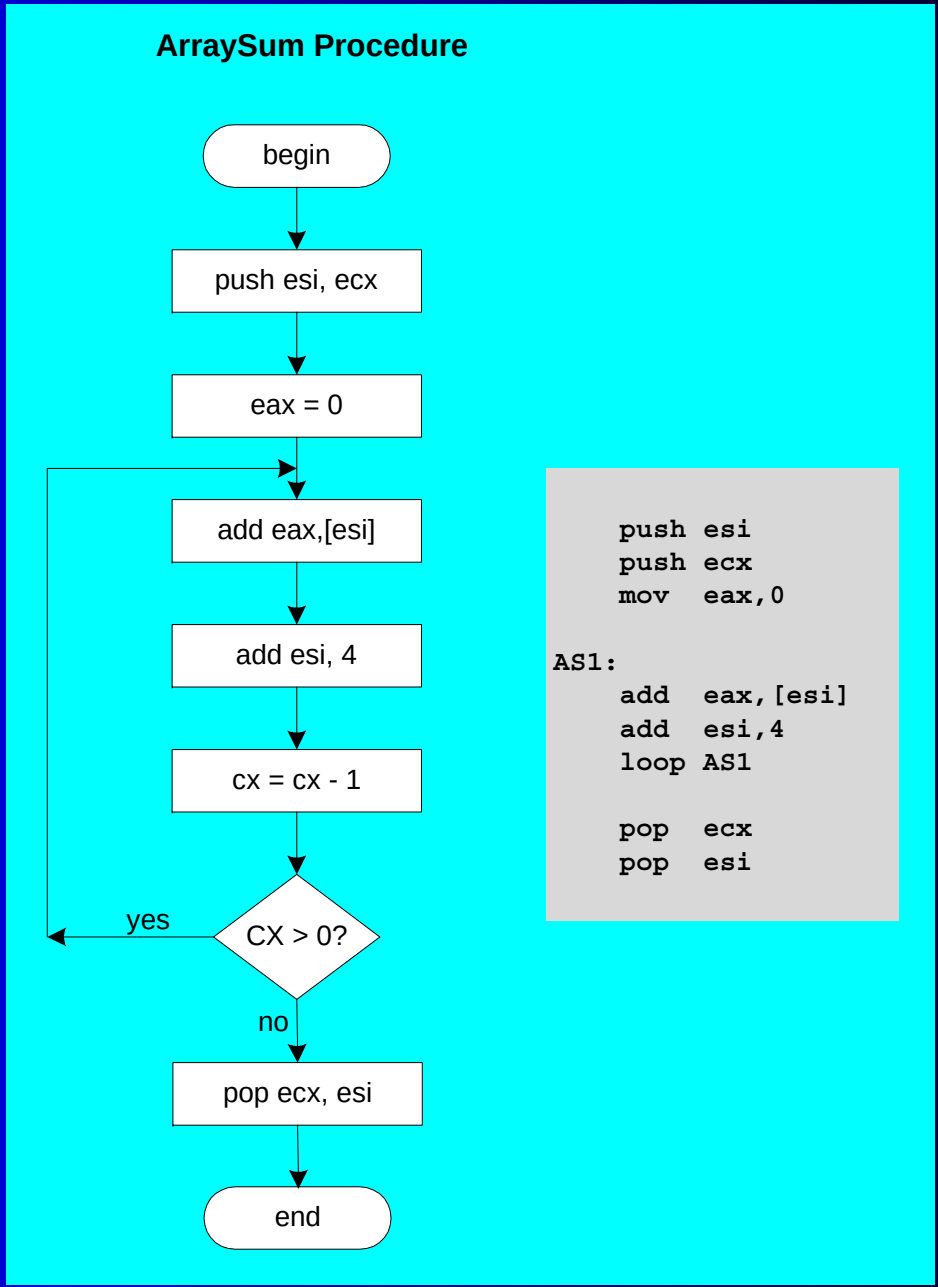
```
ArraySum PROC
; Recevies: ESI points to an array of doublewords,
;   ECX = number of array elements.
; Returns: EAX = sum
;-----------------------------------------------
    mov eax,0                   ; set the sum to zero

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 Symbols

- The following symbols are the basic building blocks of flowcharts:



Flowchart for the ArraySum Procedure



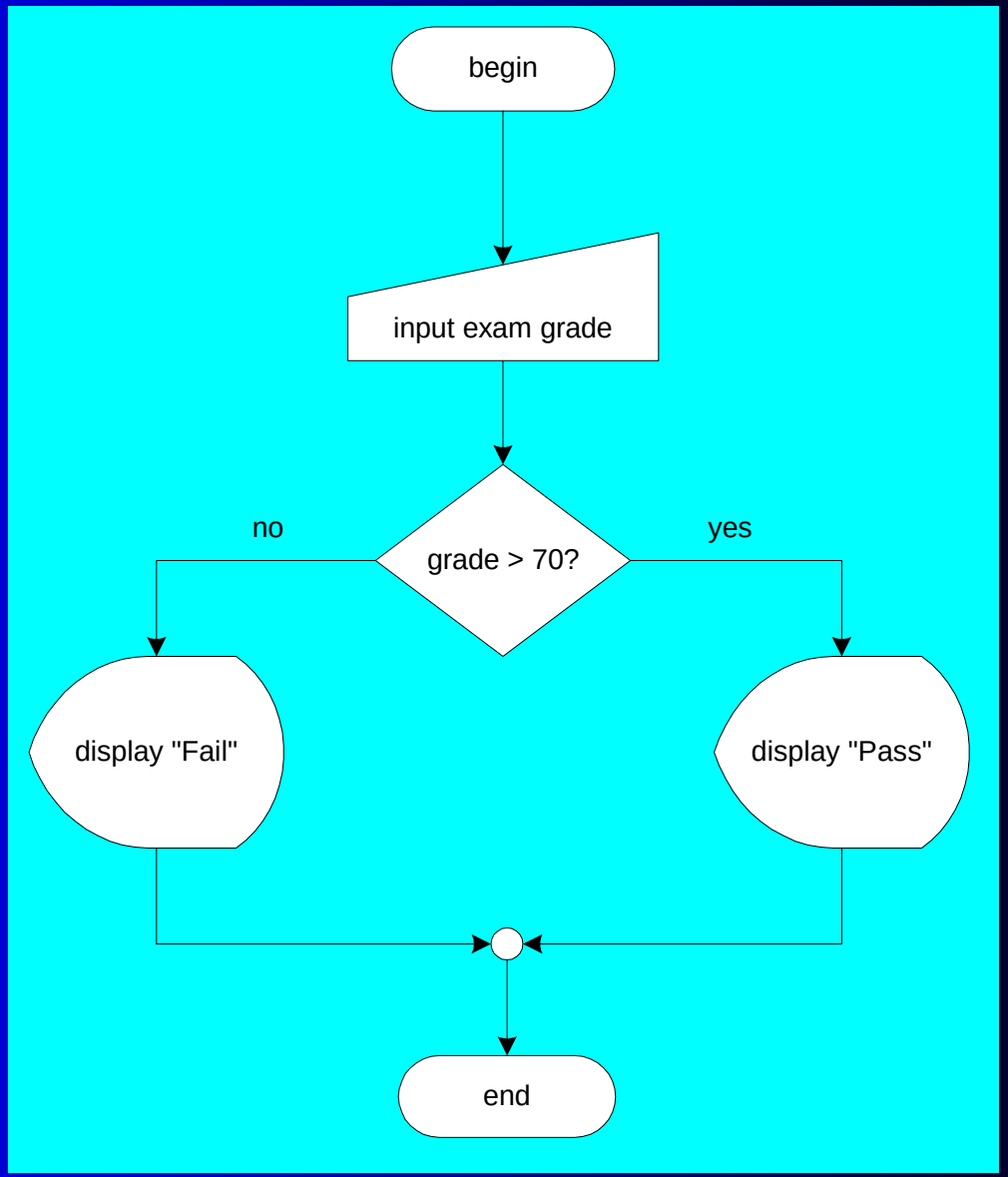

```
    push esi
    push ecx
    mov  eax,0

AS1:
    add  eax,[esi]
    add  esi,4
    loop AS1

    pop  ecx
    pop  esi
```

Your turn . . .

Draw a flowchart that expresses the following pseudocode:

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

. . . (Solution)


begin
input exam grade
grade > 70?
no
yes
display "Fail"
display "Pass"
end

Your turn . . .

- Modify the flowchart in the previous slide to allow the user to continue to input exam scores until a value of –1 is entered

USES Operator

- Lists the registers that will be saved

```
ArraySum PROC USES esi ecx
    mov eax,0                    ; set the sum to zero
    .
    .
    ret
ArraySum ENDP

; MASM generates the following code:

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
```

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 [1/4]

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.

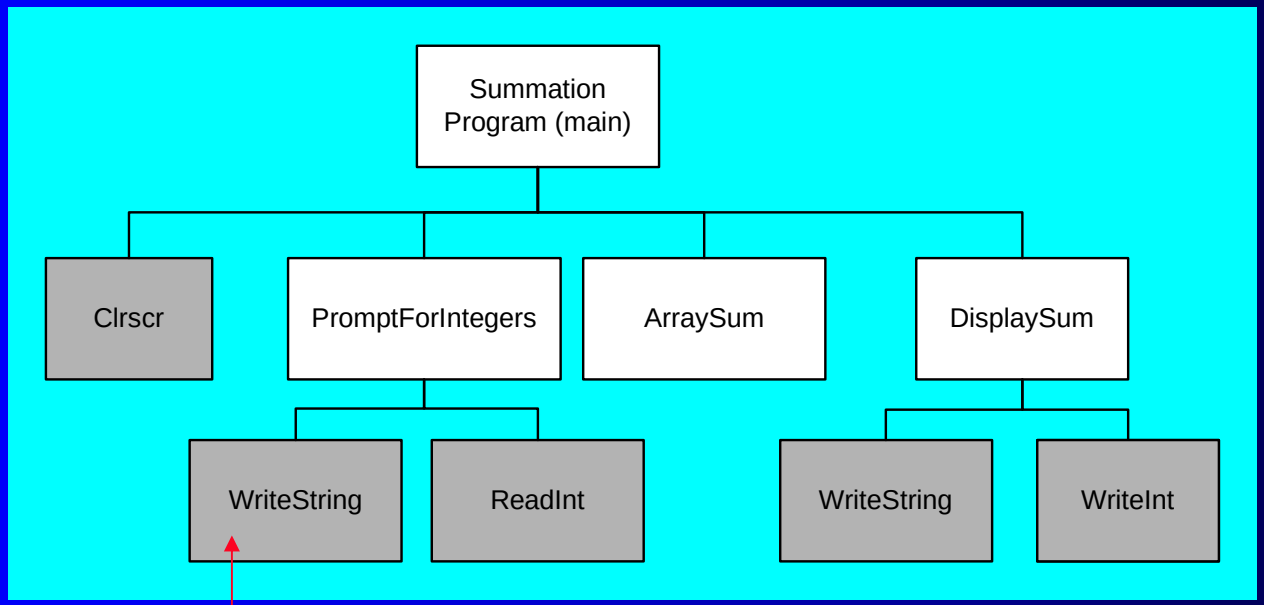
Main steps:

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

Procedure Design [2/4]

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

Structure Chart [3/4]



gray indicates library procedure

- View the stub program
- View the final program

Sample Output [4/4]

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