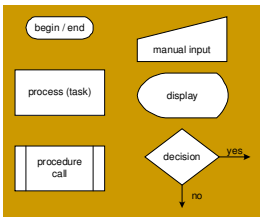
Chapter 5: Procedures

CS238 Assembly Language Programming
Amarnath Jasti

Flowchart Symbols

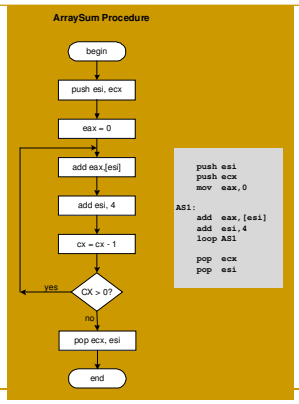
- The following symbols are the basic building blocks of flowcharts:



(Includes two symbols not listed on page 166 of the book.)

2

Flowchart for the ArraySum Procedure



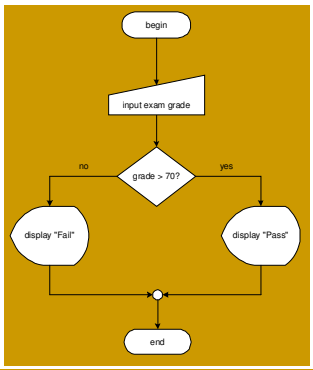
3

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
```

4



5

Int 21H

- 77 Official functions.
- Register AH defines the function.
- Examples

6

Procedures in Link Library

- Table 5-1
- Based on CALL function
- Every function predefined set of registers.

7

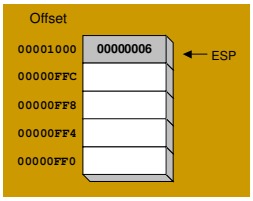
Stack Operations

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

8

Runtime Stack

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

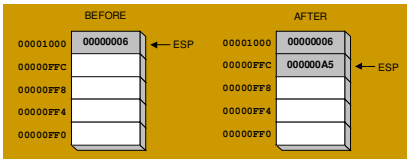


* SP in Real-address mode (32-bit)

9

PUSH Operation (1 of 2)

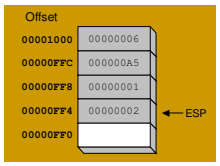
- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.



10

PUSH Operation (2 of 2)

- This is the same stack, after pushing two more integers:

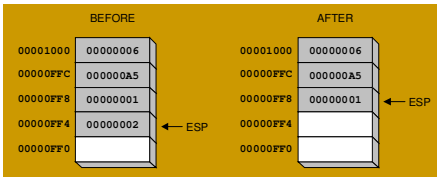


The stack grows downward. The area below ESP is always available (unless the stack has overflowed).

11

POP Operation

- Copies value at stack[ESP] into a register or variable.
- Adds *n* to ESP, where *n* is either 2 or 4.
  - depends on the attribute of the operand receiving the data



12

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. Note that the PUSH and POP instructions are in the opposite order:

```
push esi                    ; push registers
push ecx
push ebx

mov esi,OFFSET dwordVal     ; starting OFFSET
mov ecx,LENGTHOF dwordVal   ; number of units
mov ebx,TYPE dwordVal       ; size of a doubleword
call DumpMem                ; display memory

pop ebx                     ; opposite order
pop ecx
pop esi
```


Example: Nested Loop

Remember the nested loop we created on page 129? It's easy to 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
- Push each character on the stack
- Start at the beginning of the string, pop the stack in reverse order, insert each character into the string
- Q: 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.

Your turn . . .

- Using the String Reverse program as a starting point,
- #1: Modify the program so the user can input a string of up to 50 characters.
- #2: Modify the program so it inputs a list of 32-bit integers from the user, and then displays the integers in reverse order.

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

Your Turn . . .

- Write a program that does the following:
  - Assigns integer values to EAX, EBX, ECX, EDX, ESI, and EDI
  - Uses PUSHAD to push the general-purpose registers on the stack
  - Using a loop, the program pops each integer from the stack and displays it on the screen

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 of 2)

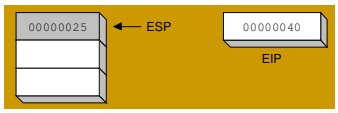
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

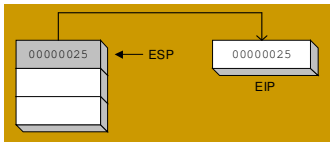
```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

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


CALL-RET Example (2 of 2)

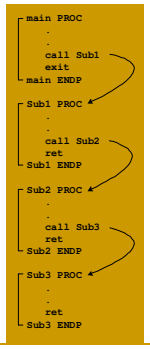
The CALL instruction pushes 00000025 onto the stack, and loads 00000040 into EIP



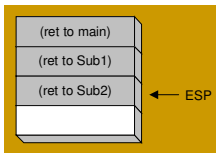
The RET instruction pops 00000025 from the stack into EIP



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.

```
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 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

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
```


USES Operator

- Lists the registers that will be saved

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

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 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.

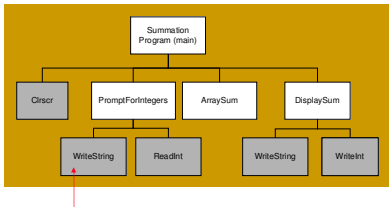
Main steps:

- 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)



gray indicates library procedure

- View the stub program
- View the final program

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
```


The End