

Lecture 6

Stack, Subroutine Instructions, Macro Directive

1

Topics

- Stack Instructions
- Subroutine Instructions
- Macro Directive
- Calling Assembly subroutine
from C Program

2

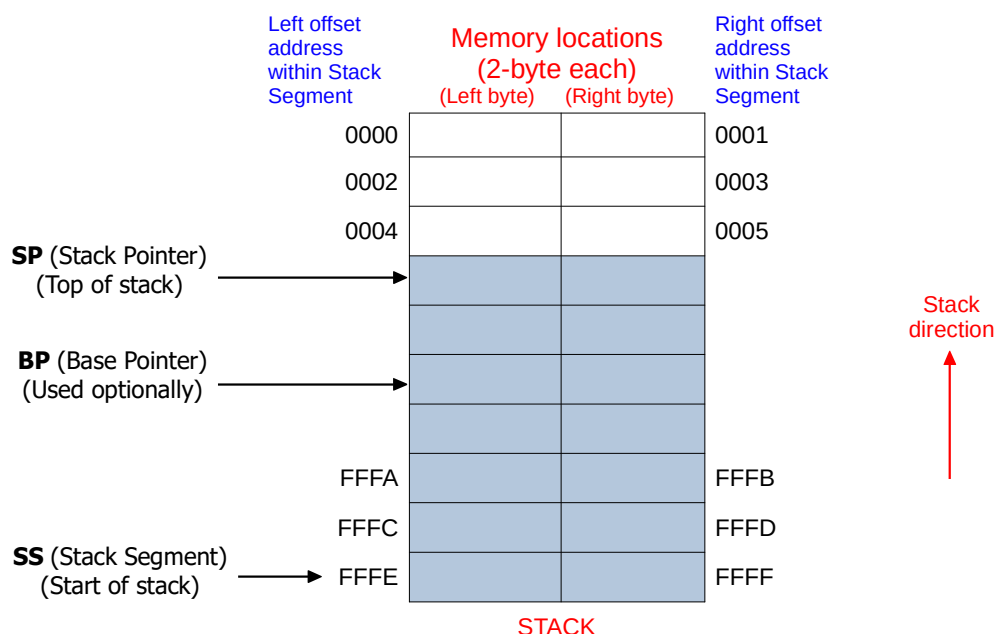
Stack Segment

- Stack Segment is a **temporary storage area** in RAM memory.
- Programmer can define the size of the Stack Segment.
Default is 1024 byte (1 KB), maximum is 64 KB.
- Stack is used for two different purposes:
 - 2-byte data can be stored and removed in stack by programmer.
 - While going to a subroutine, the **return addresses (IP=Instruction Pointer)** is stored in stack automatically by CPU.
- The beginning address of the stack is stored in Stack Segment register (SS register) by the CPU automatically.

3

Stack related registers

- Each row in the stack contains 2-byte values.
- Starting address (highest address) example: FFFFh
- Ending address (lowest address) example: 0000h
- The SP register proceeds in decreasing order.



4

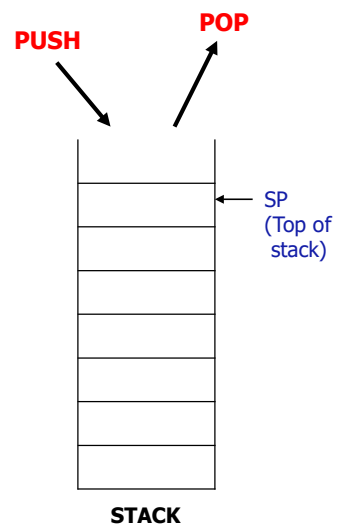
Stack related registers

- **Stack Segment register (SS) :**
 - Points to base address of segment containing the stack.
 - It is automatically initialized by CPU.
- **Stack Pointer register (SP) :**
 - Points to **top of stack** (offset address of top of stack).
 - It is automatically initialized and updated by CPU.
 - SP goes upwards, from higher addresses to lower addresses.
- **Base Pointer register (BP) :**
 - Can be used to point to start location of some data in stack.
 - It can be optionally used by programmer for Based, Based Indexed, or Register Indirect addressing modes.

5

Stack Operations

- **Push instruction (PUSH):**
 - Stores a data to top of stack.
 - Data is placed on stack first.
 - Then stack pointer (top) is decremented.
- **Pop instruction (POP) :**
 - Removes a data from top of stack.
 - Stack pointer (top) is incremented first.
 - Then data is retrieved from top of stack.



Only 16-bit (2-byte) operands can be used in 8086 stack instructions.

6

Stack Operations

- Stack is a Last-In-First-Out (**LIFO**) data structure in memory.
- Lastly added data is removed firstly.
- Push and Pop operations must be written **in reverse order**, in order **to restore data** back into its original value.
- Unbalanced PUSH and POP instructions may overwrite data in stack.

```
PUSH AX
PUSH BX
...
...
POP BX
POP AX
```

7

PUSH Instruction

General Syntax

PUSH Source

- PUSH instruction decrements the stack pointer by 2 and copies a **word (16 bits)** from a specified source to the location in the stack segment to which the stack pointer points.
- **Source of the word can be general-purpose register, segment register, or memory.**
- Stack segment register and the stack pointer must be initialized before this instruction can be used.
- PUSH can be used to save data on the stack so that it will not be destroyed by a procedure.

8

PUSH instruction examples

PUSH BX

Decrements SP (Stack Pointer) register by 2.
Then copies BX to stack.

PUSH DS

Decrements SP by 2.
Then copies DS (Data Segment) register to stack.

PUSH BL

Error: Invalid instruction. Source must be a word (2-byte).

PUSH DI[DI][BX]

Decrements SP by 2.
Then copies word from memory in DS,
at Effective Address = DI + [BX] to stack.

9

POP Instruction

General Syntax

POP Destination

- It is the opposite of PUSH instruction.
- POP instruction copies a **word (16 bit)** from the stack location pointed to by the stack pointer to a destination specified in the instruction.
- **Destination can be a general-purpose register, a segment register or a memory location.**
- Data in the stack is not changed.
- After the word is copied to the specified destination, the stack pointer (SP) is automatically incremented by 2 to point to the next word on the stack.

10

POP instruction examples

POP DX

Copies a word from top of stack to DX.
Then increments SP by 2.

POP DS

Copies a word from top of stack to DS.
Then increments SP by 2.

POP BL

Error: Invalid instruction. Destination must be a word (2-byte).

POP DI[DX]

Copies a word from top of stack to memory in DS,
with Effective Address = DI + [BX].
Then increments SP by 2.

11

Example1 : Testing the stack

Write a program for testing the stack by adding and removing some data to / from stack.

```
.model small
.stack           ; Size of stack is 1024 byte by default

.code
MOV AX, 1234h    ; 2-byte data (hexadecimal)
MOV BX, 5678h    ; 2-byte data (hexadecimal)

PUSH AX          ; Copy AX to stack
PUSH BX          ; Copy BX to stack

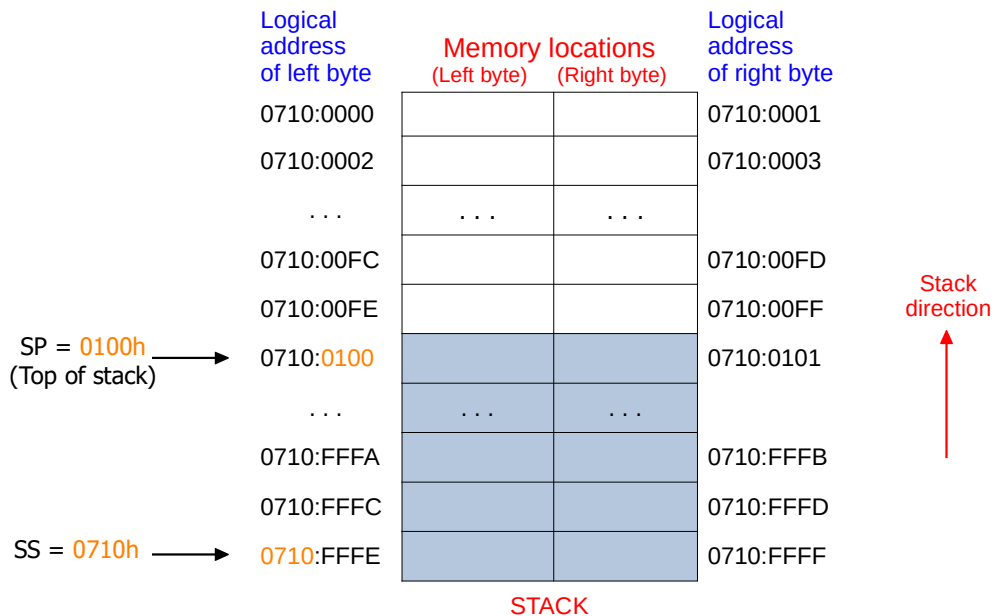
POP BX           ; Remove 2-byte from stack, copy to BX
POP AX           ; Remove 2-byte from stack, copy to AX

.EXIT
END
```

12

Initial values of stack registers

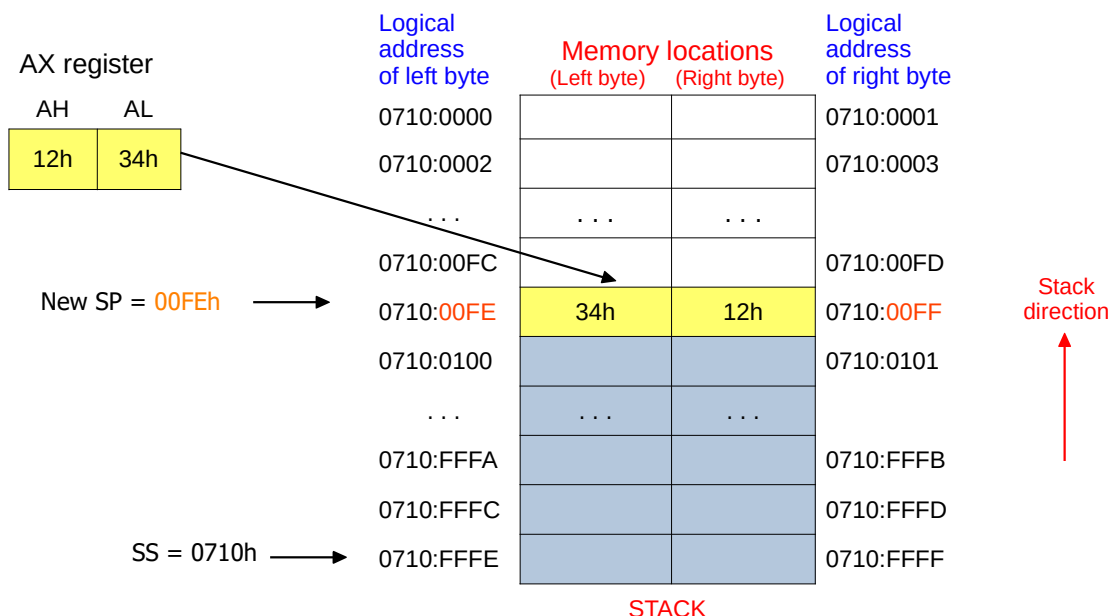
- Suppose the **SS (Stack Segment)** and the **SP (Stack Pointer)** registers are initialized by the CPU as **SS = 0710h**, **SP = 0100h**
- Physical addresses of memory locations on stack are determined by **SS:SP** logical addresses.
- The shaded rows represent the locations which are already used by CPU on the stack.



13

Execution of **PUSH AX** instruction

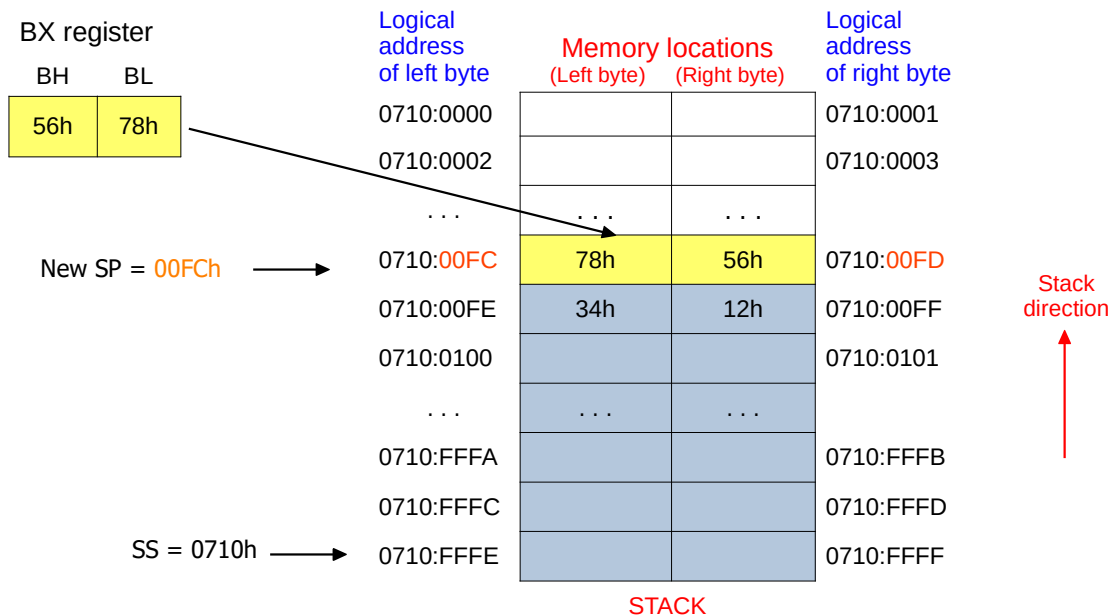
- Firstly, the SP register is decremented by 2 automatically.
- Then, AX content is copied to memory location on stack pointed by the current SP.
- Low byte of AX (34h) is stored at low address on left (0710 : 00FE).
- High byte of AX (12h) is stored at high address on right (0710 : 00FF).
- (Little Endian method is used by 8086 CPU.)



14

Execution of **PUSH BX** instruction

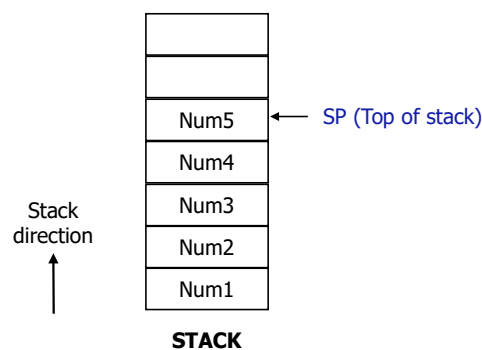
- Firstly, the SP register is decremented again by 2 automatically.
- Then, BX content is copied to memory location on stack pointed by the current SP.
- Low byte of BX (78h) is stored at low address (0710 : 00FC).
- High byte of BX (56h) is stored at high address (0710 : 00FD).



15

Example2 : Reversing an array

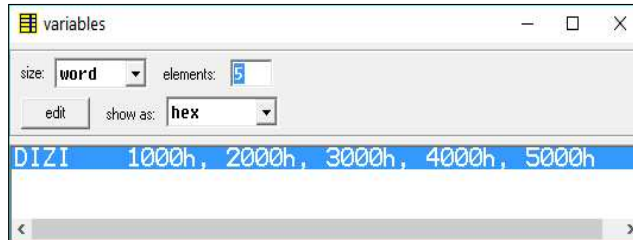
- Write a program to reverse the order of numbers in an array in memory.
- Define an **Array** of 10 elements (2-byte each) in memory.
- The reversed array should be at the same memory address.
- The stack will be used to store the numbers temporarily.
- First phase: Read all elements from Array and store (push) to Stack.
- Second phase: Read (pop) all elements from Stack and store back to Array.



16

Example : Using Emu8086 to run the program

Original array data values (hexadecimal, 2-byte each) in memory.

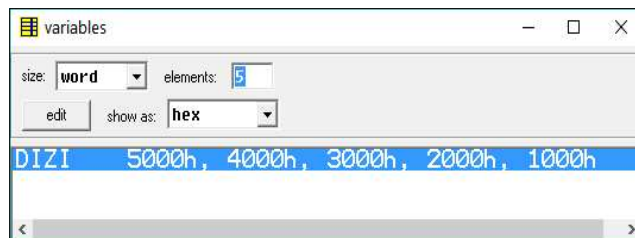


First phase

Stack Contents

0710:003E	1000
0710:003C	2000
0710:003A	3000
0710:0038	4000
0710:0036	5000 <

Reversed array data values.



Second phase

17

Program

Part1

```

;Program reverses an array, by using the stack.
;CX register is used as implicit loop counter.
.model small

.stack 65535    ;Stack size is 64 KB

.data
;Hexadecimal data values (2-byte each) in array
DIZI DW 1000h, 2000h, 3000h, 4000h, 5000h

N EQU 5        ; Constant for count of elements in array
;-----

.code
.STARTUP
;First phase: Copy data values from array to stack.
MOV BX, 0000h   ;Load array offset address to BX
MOV CX, N       ;Loop counter initialized

DONGU1:
MOV AX, DIZI [BX] ;Read next number (2-byte) from array
PUSH AX         ;Copy number from AX register to stack
ADD BX, 2       ;Increment BX by 2
LOOP DONGU1     ;Decrement CX by 1, if not zero goto loop
    
```

18

Program

Part2

;Second phase: Copy data values from stack back to array.

```
MOV BX, 0000h    ;Load array offset address to BX
MOV CX, N        ;Loop counter initialized

DONGU2:
POP AX           ;Copy number from stack to AX register
MOV DIZI [BX], AX ;Store number from AX register back to array
ADD BX, 2        ;Increment BX by 2
LOOP DONGU2      ;Decrement CX by 1, if not zero goto loop

.EXIT

END              ;End of file
```

19

PUSHF and POPF Instructions

General Syntax

PUSHF
POPF

(There are no
explicit operands)

- **PUSHF** : Push Flag Register to Stack
- **POPF** : Pop word from top of Stack to Flag Register
- These instructions do not change any status flags.
- **PUSHF** instruction decrements the stack pointer by 2 and copies a **word** in the flag register to two memory locations in stack pointed to by the stack pointer.
- Stack segment register is not changed.
-
- **POPF** instruction copies a word from two memory locations at the top of stack to the flag register and increments stack pointer by 2.
- Stack segment register and **word** on the stack are not changed.

20

Topics

- Stack Instructions
- Subroutine Instructions
- Macro Directive
- Calling Assembly subroutine from C Program

21

CALL Instruction

General Syntax

CALL operand

- **CALL** : Call a procedure (subroutine)
- A subroutine in Assembly language is similar to functions in high-level languages like C.
- The CALL instruction is used to transfer execution to a procedure (subroutine/subprogram).
- There two basic type of calls : Near and far.
- **A near call** is a call to a procedure, which is in the same code segment as the CALL instruction.

General syntax for near calling:

CALL SubroutineName

22

Procedure Definition (PROC)

- When 8086 executes a near CALL instruction, it decrements the stack pointer (SP) by 2 and copies the offset of the next instruction (IP register) after the CALL instruction into the stack.
- The offset that saved in the stack is referred to as the **return address**.
- It will also load the **instruction pointer** (IP) with the offset of the first instruction in the procedure.
- A RET instruction at the end of the procedure will return execution to the address saved on the stack which is copied back to IP.

General syntax for near subroutine definition:

```
SubroutineName PROC  
    Instructions  
    RET  
SubroutineName ENDP
```

23

RET Instruction

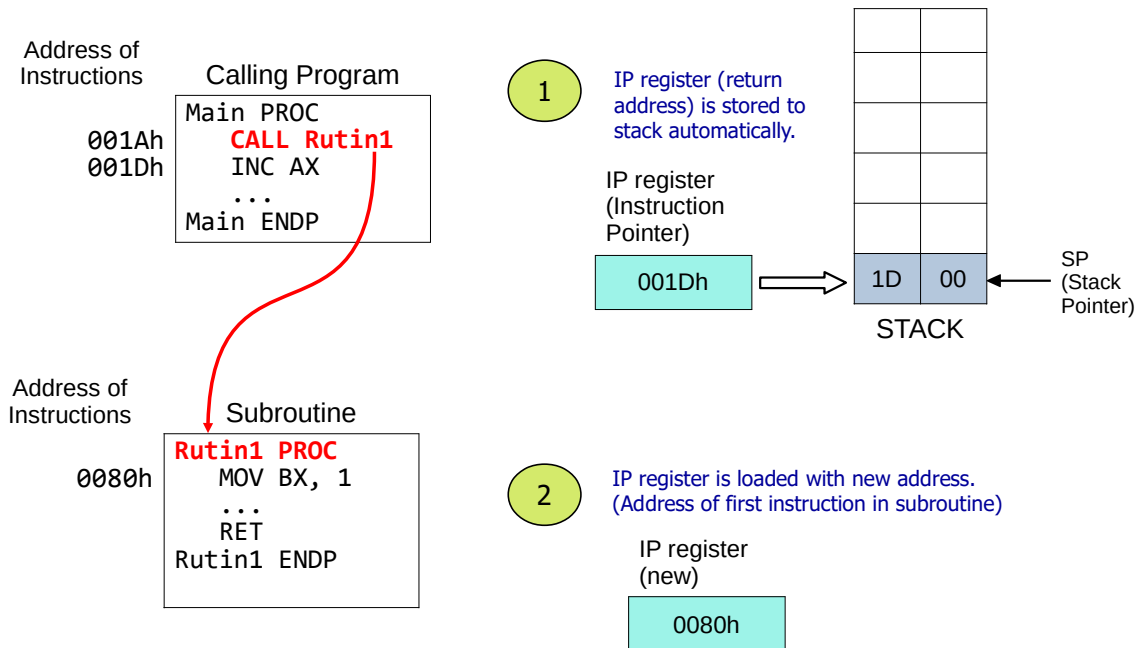
General
Syntax

RET

- **RET** : Return from a procedure
- The RET instruction is used to return execution from procedure to calling program.
- The returning location is the next instruction after the CALL instruction, which was used to call the procedure.
- **Near Procedure** : A procedure is considered as near procedure if it is in the same code segment as the CALL instruction.
The return will be done by replacing the IP register with a word from the top of stack.
The Stack Pointer register will be incremented by 2 after the return address is popped off the stack.
- **Far Procedure** : A procedure is considered as far procedure if it is in a code segment other than the one from which it is called.
Both CS register and IP register are replaced from top of the stack.

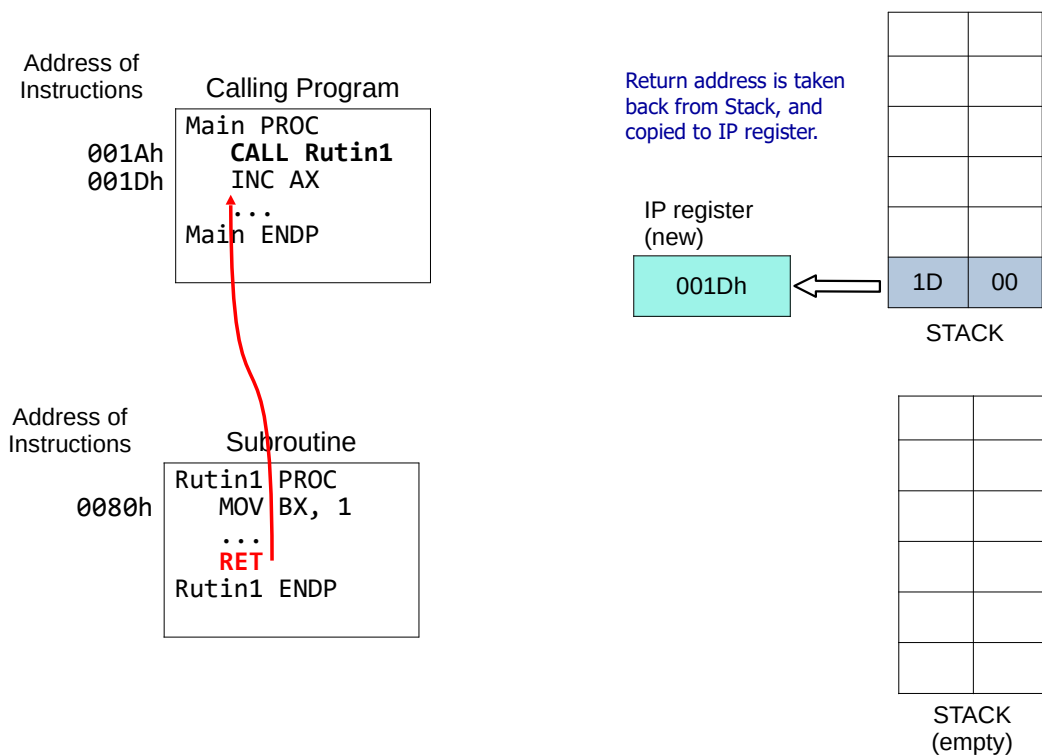
24

Calling a Near Procedure



25

Returning from a Near Procedure



26

Examples : CALL Instructions (Near calls)

Example1:

```
CALL  RUTIN1
```

- By default, the procedure call method is the NEAR call.
- Example1 is a **direct call** within same code segment (near or intra segment).
- RUTIN1 is the name of the subroutine (sub procedure).
- The Assembler determines the address displacement (offset) of RUTIN1 from the calling instruction, and uses this displacement as part of the instruction.

Example2:

```
CALL  BX
```

- Example2 is an **indirect call** within same code segment.
- BX register must contain the offset of the first instruction of the subroutine.
- It replaces the content of IP register with the content of BX register.

27

Example1 : Subroutine for Total calculation

- The main program calls a subroutine (procedure).
- Two parameters are passed to subroutine via AX and BX registers.
- Subroutine calculates the total of two numbers.
- The result is stored in AX register by the subroutine.

```
.model small
.stack                      ; Stack size is 1 KB by default.
.code
BASLA PROC                  ; Main program
    ; Copy immediate data to AX and BX registers.
    MOV AX, 1200H           ; 2-byte data
    MOV BX, 3400H           ; 2-byte data
    CALL TOPLAM_HESAPLA
    .EXIT
BASLA ENDP                  ;End of main program
;-----
TOPLAM_HESAPLA PROC        ; Procedure (Subroutine)
    ADD AX, BX              ; Add BX to AX
    RET                    ; Return from procedure
TOPLAM_HESAPLA ENDP        ; End of procedure
END Basla                  ;End of file
```

28

Example2 : Subroutine for Average calculation

- The main program calls the subroutine named **Ortalama** three times, with different data from memory.
- Two parameters are passed to subroutine via AX and BX registers.
- Subroutine calculates the average of two numbers.
- The result is stored in AX register by the subroutine.
- Main program stores the result from AX to memory.

Part1

```
.model small

.stack      ;Default stack size is 1 KB.
;The CALL instruction uses the stack to store
;the return address from a subroutine.

.data
;Variables (2-byte) are initiazlized with decimal numbers.
Num1  DW  1000
Num2  DW  2000
Num3  DW  3000

Avg_1_2  DW  ?
Avg_1_3  DW  ?
Avg_2_3  DW  ?
```

29

Part2

```
.code
BASLA PROC ;Main program
.STARTUP

;Call subroutine to calculate an average
MOV AX, Num1
MOV BX, Num2
CALL Ortalama
MOV Avg_1_2, AX ;Save average to memory

;Call subroutine to calculate an average
MOV AX, Num1
MOV BX, Num3
CALL Ortalama
MOV Avg_1_3, AX ;Save average to memory

;Call subroutine to calculate an average
MOV AX, Num2
MOV BX, Num3
CALL Ortalama
MOV Avg_2_3, AX ;Save average to memory

.EXIT
BASLA ENDP ;End of main program
```

30

Part3

```

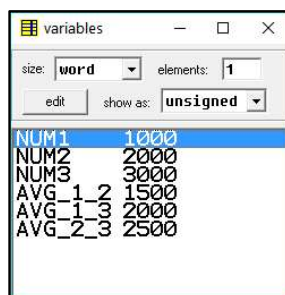
; Procedure (Subroutine)
; Implicit parameters : AX, BX registers

Ortalama PROC
    ADD AX, BX    ; Calculate total
    SHR AX, 1     ; Shift Right AX 1 time (It means divide by 2)
    RET          ; Return from procedure
                ; Result (average) is in AX
Ortalama ENDP    ; End of subroutine

END Basla        ; End of file

```

Calculated averages in
Emu8086 memory variables.



Variable	Value
NUM1	1000
NUM2	2000
NUM3	3000
AVG_1_2	1500
AVG_1_3	2000
AVG_2_3	2500

31

Example3 : Using the **INCLUDE** directive for file inclusion

- Some parts of a long Assembly program can be saved in several files separately.
- In main program, other source files can be included in main file, by using the INCLUDE Assembler directive.

Main.asm (Program file)

```

.model small
.stack
.data

INCLUDE dosya.inc
;Data segment variables are
;included from a separate file.

.code
BASLA PROC
.STARTUP

; WRITE MAIN PROGRAM CODES

END Basla ;End of file

```

dosya.inc File

```

;This text file can be included
; by other Assembly programs.

Num1 DW 1000
Num2 DW 2000
Num3 DW 3000

Avg_1_2 DW ?
Avg_1_3 DW ?
Avg_2_3 DW ?

```

32

Topics

- Stack Instructions
- Subroutine Instructions
- Macro Directive
- Calling Assembly subroutine from C Program

33

Macro Directive

- A macro is a function that can be called from a main program.
- It is similar to an inline-function used in high-level languages such as C.
- During the compiling process, the Assembler (compiler) substitutes (copy+paste) the block of statements of macro, whenever it is called at various locations in main program.
- Parameters of macro are optional.

Syntax for macro declaration:

```
Name-of-macro MACRO parameter1, parameter2, ...  
  
    Instructions  
  
ENDM
```

34

Example : Macro for Average calculation

- The following macro takes 3 parameters.
2 source parameters, and 1 target parameter.
- The macro calculates the average of source numbers, and copies result to target.
- The macro will be called (invoked) several times from the main program.

Part1

```
; Macro function
; Macro name: Ortalama
; Parameters (arguments) : Kaynak1, Kaynak2, Hedef

Ortalama MACRO Kaynak1, Kaynak2, Hedef
; Calculate the average, then save it to the target.
MOV AX, Kaynak1 ; Copy first parameter
MOV BX, Kaynak2 ; Copy second parameter
ADD AX, BX      ; Calculate total
SHR AX, 1       ; Shift Right AX 1 time (It means divide by 2)
                ; Result (average) is in AX
MOV Hedef, AX   ; Save average to target
ENDM          ; End of macro
```

35

Part2

```
.model small

.data
Num1 DW 1000
Num2 DW 2000
Num3 DW 3000

Avg_1_2 DW ?
Avg_1_3 DW ?
Avg_2_3 DW ?

;-----
.code
BASLA PROC ; Main program
.STARTUP
;Calling (invoking) the macro function Ortalama three times.
Ortalama Num1, Num2, Avg_1_2
Ortalama Num1, Num3, Avg_1_3
Ortalama Num2, Num3, Avg_2_3
.EXIT
BASLA ENDP ; End of Main program

END Basla ;End of file
```

36

Differences between Procedures and Macros

Basis	Procedure (Subroutine)	Macro (Inline function)
Assemble process	A procedure is a single section of code that is called from various points in the program.	A macro is a sequence of instructions that Assembler replicates in the program each time the macro is called.
Execution Speed	Execution of a procedure is usually faster. Automatic usage of stack makes procedure slower due to call/return sequence.	Execution of a macro expansion is usually faster. Macros execute faster by eliminating the call/return sequence.
Assembled machine code size	Usually programs using procedures will have smaller file size (assembled executable machine code.)	Assembler copies the macro code into the program at each macro invocation. If there are a lot of macro invocations within the program, the assembled executable machine code file will be much larger than the same program that uses procedures.

37

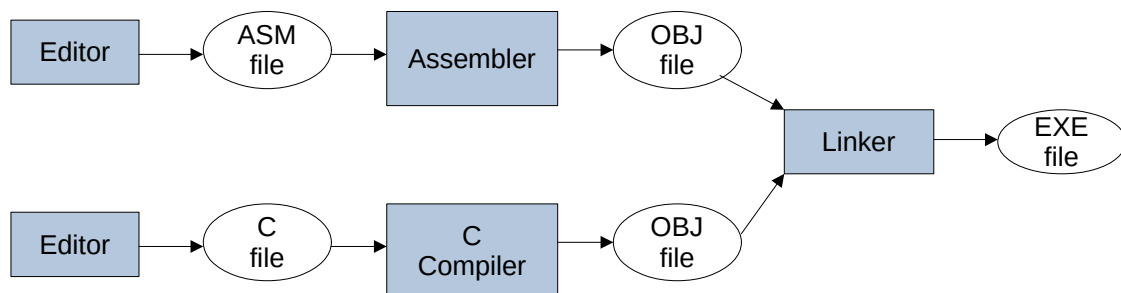
Topics

- Stack Instructions
- Subroutine Instructions
- Macro Directive
- Calling Assembly subroutine from C Program

38

Calling an Assembly subroutine from a C Program

- An Assembly language subroutine (procedure) can be called from any high-level program such as C or C++.
- Firstly, all of the source program files are compiled separately, generating the Object files (*.OBJ) for each source file.
- Then, a linker program is used to link all object files, generating one executable file (*.EXE) which contains one binary machine code file.



39

Example1 : C Program for Calculating **sum** of two numbers

- The C program below calls the **Toplam** function.
- The Toplam function is an Assembly language subroutine (procedure).

TOTMAIN.C
File

```
#include <stdio.h>
int main()
{
    // Calling the function
    printf("Total = %d \n", Toplam(20, 30) );
}
```

- Assembly procedure takes two input parameters and calculates the total.
- The result is returned to the C program, then displayed on console screen.
- All commands for compiling and linking should be entered in **DOSBox emulator**.
- **MASM Assembler** (version 6.11) is used for compiling the Assembly program.
- **Borland Turbo C compiler and linker** (version 2.01) is used for compiling the C program, and for linking the object files.

40

Step1) Compiling the Assembly subroutine

- The Assembly subroutine named Toplam retrieves two parameters from the Stack.
- First, the Base Pointer (BP) register is stored in stack for backup purpose.
- Then, Stack Pointer (SP) register is copied to BP register.
- The calculated result (total of numbers) is stored in AX register, which is retrieved automatically by the C compiler in the C program.

TOTAL.ASM
File

```
.model small
.code
; Underscore prefix required by Linker as the C naming convention.

_Toplam PROC
    PUSH BP          ;Save (backup) Base Pointer register into stack
    MOV BP, SP       ;Copy Stack Pointer register to BP
    MOV AX, [BP+4]    ;Retrieve first Argument from BP+4 on stack
    ADD AX, [BP+6]    ;Retrieve second Argument from BP+6 on stack
    POP BP           ;Restore Base Pointer register from stack
    RET              ;Return from subroutine (uses BP)
_Toplam ENDP

END                  ;End of file
```

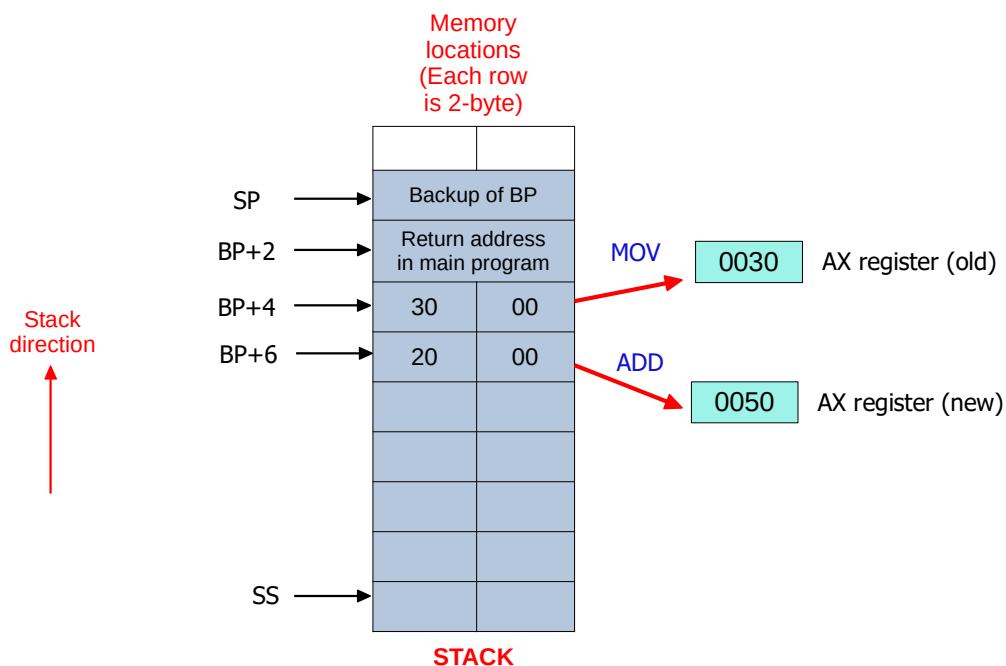
- MASM Assembler is used for compiling.
- The /c option means compile only.
- The TOTAL.OBJ file is generated.

```
C:> ML.EXE /c TOTAL.ASM
```

41

Usage of Stack in the Assembly subroutine

- From location [BP+4] on stack, the first parameter is retrieved.
- From location [BP+6] on stack, the second parameter is retrieved.



42

Step2) Compiling the C Main program, and Linking with the Assembly subroutine

- The TCC.EXE program is the command-line Turbo C compiler and linker.
- It generates TOTMAIN.OBJ file and also the TOTMAIN.EXE.
- Linker links the TOTMAIN.OBJ file (C) and the TOTAL.OBJ file (Assembly).
- By default, linker output is the TOTMAIN.EXE file.

```
C:> TCC.EXE TOTMAIN.C TOTAL.OBJ
```

Step3) Running the Program

```
C:> TOTMAIN.EXE  
Total = 50
```

43

Example2 : C Program for Calculating **average** of two numbers

- The C program below calls Assembly subroutine **Ortalama** to calculate average of two numbers.
- Main C program reads the numbers from keyboard entered by user.

AVGMAIN.C
File

```
#include <stdio.h>

void main()
{
    int Num1, Num2, Sonuc;
    printf ("Enter two integers : ");
    scanf ("%d %d", &Num1, &Num2);
    Sonuc = Ortalama (Num1, Num2);
    printf ("Average = %d \n", Sonuc);
}
```

44

- In the Assembly subroutine named **Ortalama**, the C calling syntax is used.
 PROC keyword : Procedure
 C keyword : C calling convention
 SWORD keyword : Signed Word (data type of parameters)

AVERAGE.ASM File

```
.MODEL small, C ; C keyword declares the C calling convention
.CODE

Ortalama PROC C number1 : SWORD , number2 : SWORD
    MOV AX, number1 ; Load Argument1 into AX
    MOV BX, number2 ; Load Argument2 into BX
    ADD AX, BX       ; Add two numbers
    MOV BX, 2        ; For division
    IDIV BX          ; Integer Division (signed)
                    ; Divide AX (implicit operand) by 2 (for average)
    MOV AH, 0        ; Clear AH (remainder of division)
    RET              ; Return (Result is in AL as part of AX)
Ortalama ENDP       ; End of Procedure

END                 ; End of file
```

45

Compiling and Linking Steps

Step1) MASM assembler generates the AVERAGE.OBJ file.

```
C:> ML.EXE /c AVERAGE.ASM
```

Step2) Turbo C Compiler generates the AVGMAIN.OBJ file (C).
 Linker takes the AVGMAIN.OBJ file (C) and the AVERAGE.OBJ file (Assembly) as inputs, and generates the AVGMAIN.EXE as output file.

```
C:> TCC.EXE AVGMAIN.C AVERAGE.OBJ
```

Step3) Running the program.

```
C:> AVGMAIN.EXE
Enter two integers : 20 50
Average = 35
```

46