

LAB 07

Computer Organization & Assembly Language (COAL) | EL 2003



Student Name: _____

Roll No.: _____

Section: _____

Marks Awarded _____

Prepared by: Talha Shahid

Objectives:

- Stack and Runtime Stack
- Push & Pop Operations
- `push` and `pop` Instructions
- Procedures
- `PROC` Directive
- `CALL` & `RET` Instructions
- Nested Procedures Calls

Section 01 | Stack and Runtime Stack

- LIFO (Last-In, First-Out) data structure.
- `push`/`pop` operations
- You probably have had experiences on implementing it in high-level languages.
- Here, we concentrate on runtime stack, directly supported by hardware in the CPU. It is essential for calling and returning from procedures.

Runtime Stack

The runtime stack is a memory array managed directly by the CPU, using the `ESP` register, known as the stack pointer register. The `ESP` register holds a 32-bit `offset` into some location on the stack. We rarely manipulate `ESP` directly; instead, it is indirectly modified by instructions such as `CALL`, `RET`, `PUSH`, and `POP`.

`ESP` always points to the last value to be added to, or pushed on, the top of stack. To demonstrate, let's begin with a stack containing one value. In Figure 01 the `ESP` (extended stack pointer) contains hexadecimal `00001000`, the `offset` of the most recently pushed value (`00000006`). In our diagrams, the top of the stack moves downward when the stack pointer decreases in value:

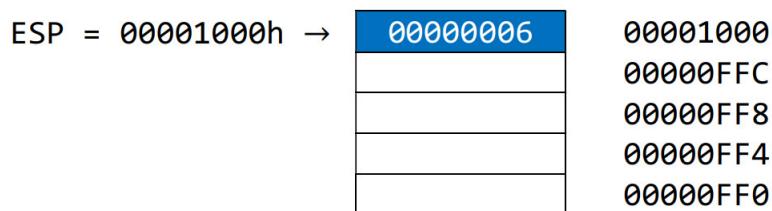


Figure 01: Runtime Stack

Each stack location in this figure contains 32 bits, which is the case when a program is running in 32-bit mode. In 16-bit real-address mode, the [SP](#) register points to the most recently pushed value and stack entries are typically 16 bits long.

Section 02 | Push & Pop Operations

Push Operation

A 32-bit push operation decrements the stack pointer by [4](#) and copies a value into the location in the stack pointed to by the stack pointer. Figure 02 shows the effect of pushing [000000A5](#) on a stack that already contains one value ([00000006](#)). Notice that the [ESP](#) register always points to the top of the stack. The figure shows the stack ordering opposite to that of the stack of plates we saw earlier, because the runtime stack grows downward in memory, from higher addresses to lower addresses. Before the push, [ESP = 00001000h](#); after the push, [ESP = 00000FFCh](#). Figure 03 shows the same stack after pushing a total of four integers.

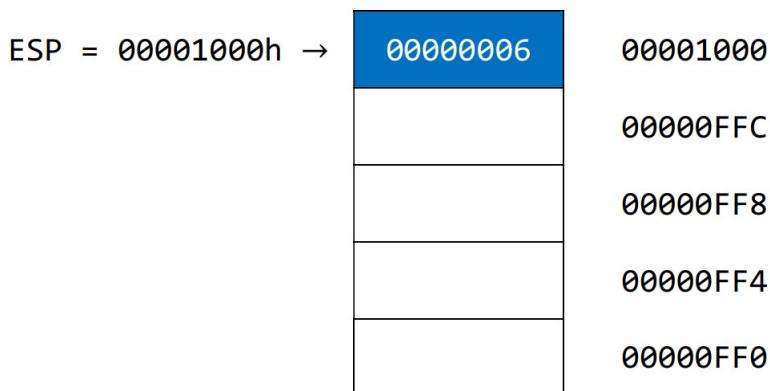


Figure 02: Before

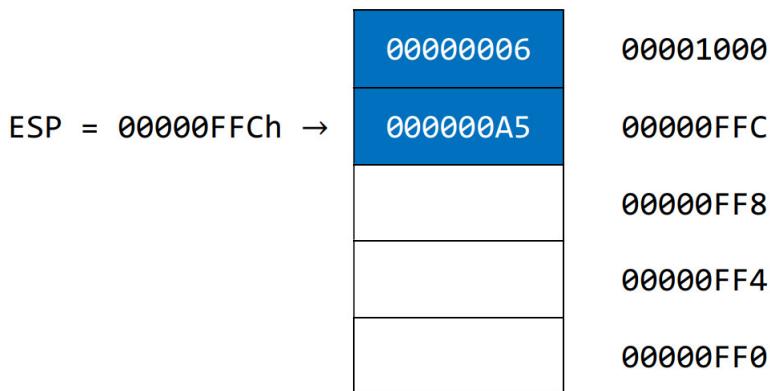


Figure 03: After

Pop Operation

A pop operation 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. Figure 04 & 05 shows the stack before and after the value `00000002` is popped.

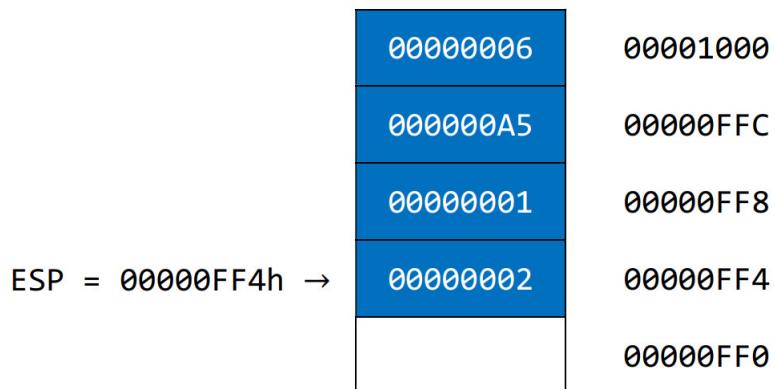


Figure 04: Before

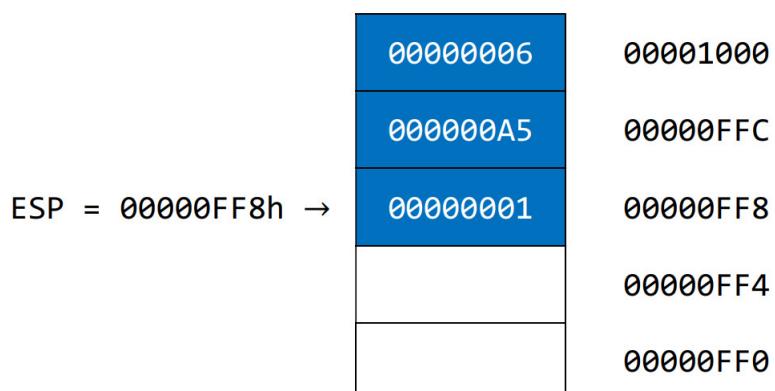


Figure 05: After

The area of the stack below `ESP` is logically empty, and will be overwritten the next time the current program executes any instruction that pushes a value on the stack.

Section 03 | push & pop Instructions

PUSH Instruction

The **PUSH** instruction first decrements **ESP** and then copies a source operand into the stack. A 16-bit operand causes **ESP** to be decremented by **2**. A 32-bit operand causes **ESP** to be decremented by **4**. There are three instruction formats:

PUSH reg/mem16

PUSH reg/mem32

PUSH imm32

Immediate values are always 32 bits in 32-bit mode.

POP Instruction

The **POP** instruction first copies the contents of the stack element pointed to by **ESP** into a 16- or 32-bit destination operand and then increments **ESP**. If the operand is 16 bits, **ESP** is incremented by **2**; if the operand is 32 bits, **ESP** is incremented by **4**:

POP reg/mem16

POP reg/mem32

Example: Displays the product of three integers through a stack

```
INCLUDE Irvine32.inc
.data
    multp DWORD 2
.code
main PROC
    mov eax, 1
    mov ecx, 3

    L1:
        push multp
        add multp, 2
        loop L1

    mov ecx, 3

    L2:
        pop ebx
        mul ebx ;eax value multiply
        loop L2

    call WriteDec
```

```
    exit
main ENDP
END main
```

Example: To find the largest number through a stack

```
INCLUDE Irvine32.inc

.code
main PROC
    push 5
    push 7
    push 3
    push 2
    mov eax, 0 ;eax is the largest
    mov ecx, 4

    L1:
        pop edx
        cmp edx, eax
        jl SET
        mov eax, edx
    SET:
        loop L1

    call WriteDec
    exit
main ENDP
END main
```

Section 04 | Procedures

- Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size.
- Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well-defined job.
- End of the procedure is indicated by a return statement.

Section 05 | PROC Directive

We can define a procedure as a named block of statements that ends in a return statement. A procedure is declared using the **PROC** and **ENDP** directives. It must be assigned a name (a valid identifier). When we create a procedure other than your program's startup procedure, end it with a **RET** instruction. **RET** forces the **CPU** to return to the location from where the procedure was called:

Let's say, **sample** is the name of procedure:

```
sample PROC
    .
    .
    ret
sample ENDP
```

The procedure is called from another function by using the **CALL** instruction. The **CALL** instruction should have the name of the called procedure as an argument as shown below.

CALL Sample

The called procedure returns the control to the calling procedure by using the **RET** instruction.

Section 06 | CALL & RET Instructions

CALL Instruction

CALL instruction is used whenever we need to make a call to some procedure or a subprogram. Whenever a **CALL** is made, the following process takes place inside the microprocessor:

- The address of the next instruction that exists in the caller program (after the program **CALL** instruction) is stored in the stack.
- The instruction queue is emptied for accommodating the instructions of the procedure. Then, the contents of the instruction pointer (**IP**) are changed with the address of the first instruction of the procedure.
- The subsequent instructions of the procedure are stored in the instruction queue for execution.
- The Syntax for the **CALL** instruction is mentioned above.

RET Instruction

RET instruction stands for return. This instruction is used at the end of the procedures or the sub-programs. This instruction transfers the execution to the caller program. Whenever the RET instruction is called, the following process takes place inside the microprocessor:

- The address of the next instruction in the mainline program which was previously stored inside the stack is now again fetched and is placed inside the instruction pointer (IP).
- The instruction queue will now again be filled with the subsequent instructions of the mainline program.

Example 01:

```
INCLUDE Irvine32.inc
.data
    var1 DWORD 5
    var2 DWORD 6
.code
    main PROC
        call AddTwo
        call WriteInt
        call crlf
    exit
    main ENDP

    AddTwo PROC
        mov eax,var1
        mov ebx,var2
        add eax,var2
        ret
    AddTwo ENDP
END main
```

Example 02:

```
INCLUDE Irvine32.inc
    INTEGER_COUNT = 3

.data
    str1 BYTE "Enter a signed integer: ",0
    str2 BYTE "The sum of the integers is: ",0
    array DWORD INTEGER_COUNT DUP(?)
.code
    main PROC
        call Clrscr
        mov esi,OFFSET array
        mov ecx,INTEGER_COUNT
```

```

    call PromptForIntegers
    call ArraySum
    call DisplaySum
    exit
main ENDP

PromptForIntegers PROC USES ecx edx esi

    mov edx,OFFSET str1 ; "Enter a signed integer"

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,TYPE DWORD ; next integer
    loop L1
    ret
PromptForIntegers ENDP

ArraySum PROC USES esi ecx

    mov eax,0 ; set the sum to zero

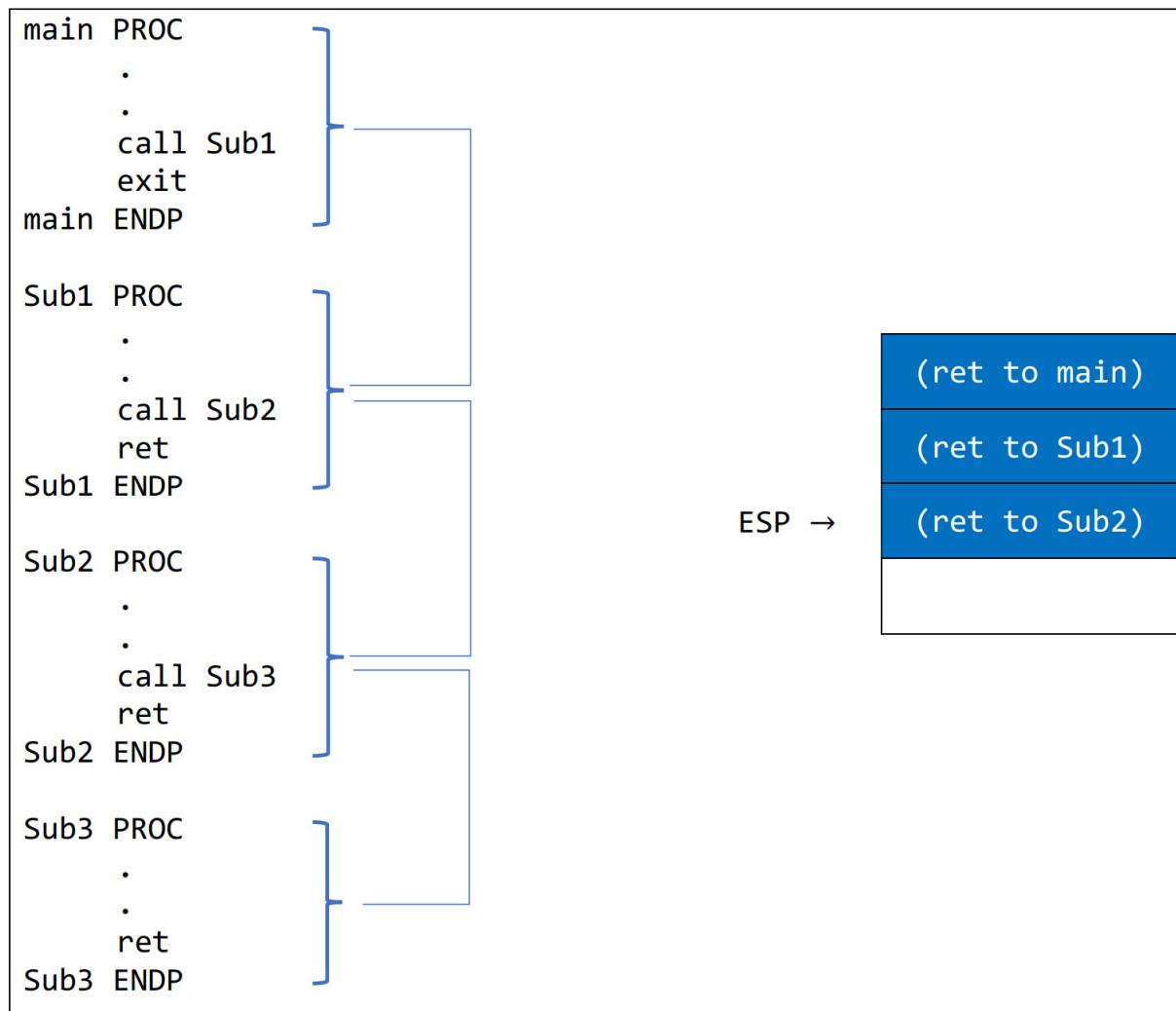
L1:
    add eax,[esi] ; add each integer to sum
    add esi,TYPE DWORD ; point to next integer
    loop L1 ; repeat for array size
    ret ; sum is in EAX
ArraySum ENDP

DisplaySum PROC USES edx
    mov edx,OFFSET str2
    call WriteString
    call WriteInt ; display EAX
    call Crlf
    ret
DisplaySum ENDP
END main
AddTwo PROC
    mov eax,var1
    mov ebx,var2
    add eax,var2
    ret
AddTwo ENDP
END main

```

Section 07 | Nested Procedures Calls

A nested procedure call occurs when a called procedure calls another procedure before the first procedure returns.



Example:

```
INCLUDE Irvine32.inc

.data
    var1 DWORD 5
    VAR2 DWORD 6

.code
    main PROC

        call AddTwo
        call WriteInt
        exit
    main ENDP
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (NUCES), KARACHI

```
AddTwo PROC
    mov eax, var1
    mov ebx, var2
    add eax, var2
    call AddTwo1
    ret
AddTwo ENDP

AddTwo1 PROC
    mov ecx, var1
    mov edx, var2
    add ecx, var2
    call WriteInt
    call Crlf
    ret
AddTwo1 ENDP
END main
```