Stack and Procedures Operations

Contents of Lecture

- Stack
- Defining and Using Procedures
- **EXERCISES**

References for this lecture

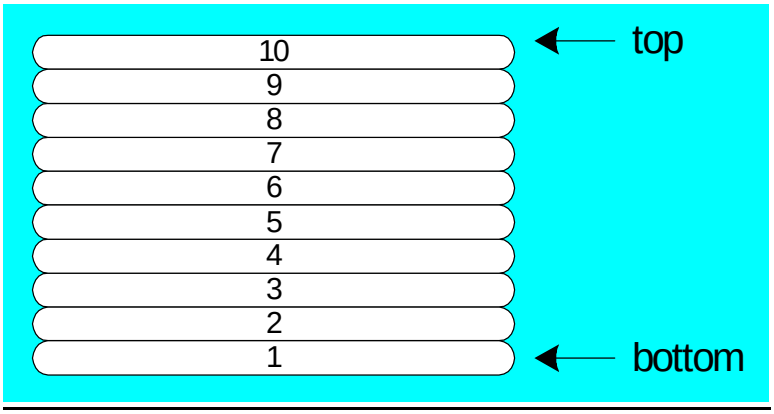
Assembly Language for x86 Processors (7th Edition). Chapter 5, Procedures

Stack:

- A stack data structure follows the same principle as a stack of plates: New values are added to the top of the stack, and existing values are removed from the top.

- A stack is also called a LIFO structure (Last-In, First-Out) because the last value put into the stack is always the first value taken out.

- Following figure show Stack of plates.

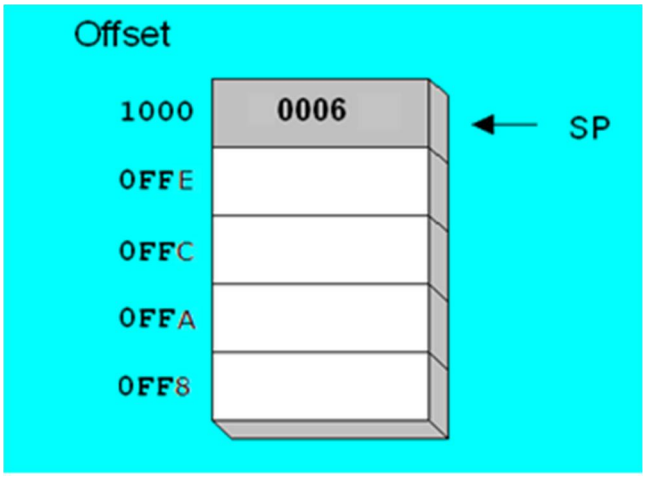


Runtime Stack

- The runtime stack is a memory array managed directly by the CPU, using the **SP** register, known as the stack pointer register.
- The **SP** register holds a 16-bit offset into some location on the stack. **SP** always points to the last value to be added to or pushed on the top of stack.

- We rarely manipulate **SP** directly; instead, it is indirectly modified by instructions such as CALL, RET, PUSH, and POP.

- To demonstrate, let's begin with a stack containing one value. In next Figure, the **SP** (stack pointer) contains hexadecimal 1000, the offset of the most recently pushed value (0006).In our diagrams, the top of the stack moves downward when the stack pointer decreases in value



- Each stack location in above figure contains 16 bits, which is the case when a program is running in 16-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.

PUSH Operation:

- A 16-bit push operation decrements the stack pointer by 2 and copies a value into the location in the stack pointed to by the stack pointer.

- Next figure shows the effect of pushing 00A5 on a stack that already contains one value (0006).
- ***Notice that*** the SP register always points to the top of the stack.
- Before the push, SP = 1000h; after the push, SP = 0FFEh.
- Next figure shows the same stack after pushing a total of four integers.

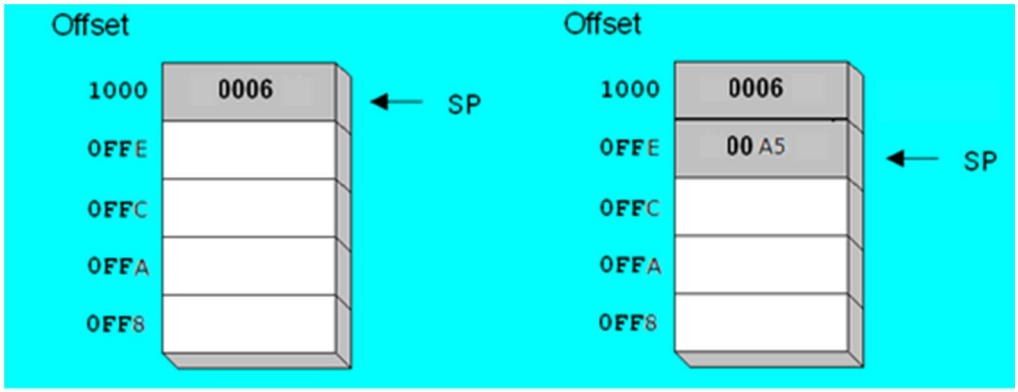
- **Syntax:**

PUSH source-16bits

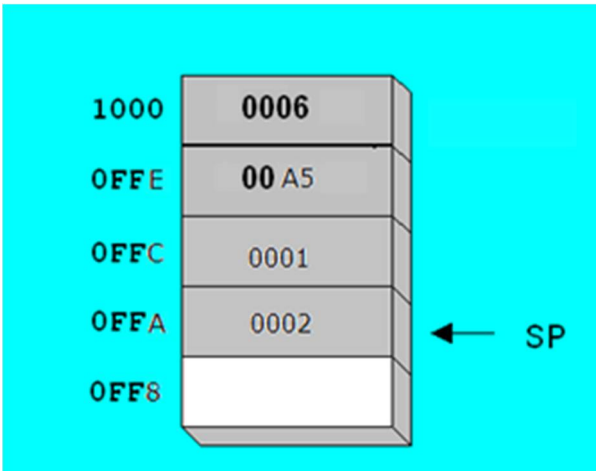
Where:

PUSH reg/mem16
PUSH reg/mem32
PUSH imm32

- A 16-bit push operation copies a value from source variable or register into the location pointed to by the stack pointer then decrements the stack pointer by 2



- Stack, after Pushing 0001 and 0002



POP Operation:

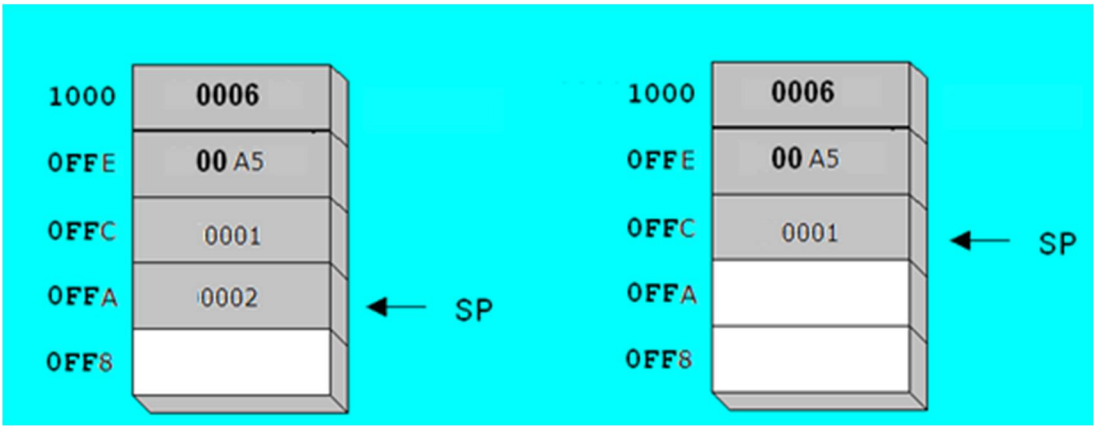
- A pop operation removes a value from the stack.
- After the value is popped from the stack, the stack pointer is incremented (by the stack element size) to point to the next highest location in the stack.
- Next figure shows the stack before and after the value 0002 is popped.
- **Syntax:**

POP destination-16bits

Where:

POP reg/mem16
POP reg/mem32

- A 16-bit pop operation copies a value into the destination variable or register then incremented stack pointer by 2.



Example:

By using stack write a program asks user to enter string of letter until press enter, program print the string inverse.

Code:

Defining and Using Procedures:

- It is useful to divide programs into subroutines. A complicated problem is usually divided into separate tasks before it can be understood, implemented, and tested effectively.

- In assembly language, we typically use the term procedure to mean a subroutine. In other languages, subroutines are called methods or functions.

Creating Procedures

- 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 you 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

- **Syntax:**

```
P-name PROC
        Statement/s
        ret
P-name ENDP
```

- Following is an assembly language procedure named sample:

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

- Each program contains a procedure named main, main procedure can contain RET or not:

```
main PROC
.
.
ret
main ENDP
```

CALL and RET Instructions:

- The CALL instruction calls a procedure by directing the processor to begin execution at a new memory location.
- The procedure uses a RET (return from procedure) instruction to bring the processor back to the point in the program where the procedure was called.
- The CALL instruction pushes its return address on the stack and copies the called procedure's address into the instruction pointer.
- When the procedure is ready to return, its RET instruction pops the return address from the stack into the instruction pointer.
- In 16-bit mode, the CPU executes the instruction in memory pointed to by IP (instruction pointer register).
- Main That; the CALL instruction calls a procedure by
  - ✓ pushes offset of next instruction on the stack
  - ✓ copies the address of the called procedure into IP
- And; the RET instruction returns from a procedure
  - ✓ pops top of stack into IP

Example:

By using procedure and stack write a program asks user to enter string of letter until press enter, program print the string inverse.

Code:

```
.model small
.stack
.data

.code
main proc
            mov   ax,@data
            mov   ds,ax

            call  p

            mov   ah,4ch
            int   21h
main endp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
p PROC
        xor     cx,cx
        top:
                mov     ah,1h
                int     21h
                cmp     al,0dh
                 je     pri

                push    ax
                inc     cx
                jmp     top
        pri:
                 mov    ah,2h
                 mov    dl,0dh
                int     21h
                mov     dl,0ah
                int     21h
        print:
                pop     dx
                mov     ah,2h
                int     21h
                 loop   print
        ret
p endp
end
```

EXERCISES

(Write the following programs by using procedure and logical operations)

1. Write a program ask user to enter a character, then the program print the ASCII code for the character in binary form at the next line and print the number of bits contain the number 1.

Example:

```
TYPE  A  CHARACTER     :    A
THE ASCII   CODE  OF A IN BINARY IS   01000001
THE NUMBER OF 1 BITS IS  2
```

2. Write a program ask user to enter a character, then the program print the ASCII code for the character in hexadecimal form at the next line. Then the program ask user again, again and again until user press enter without enter a character.

Example:

TYPE A CHARACTER : 7
THE ASCII CODE OF 7 IN HEX IS : 37
TYPE A CHARACTER : A
THE ASCII CODE OF A IN HEX IS : 41
TYPE A CHARCTER :

3. Write a program ask user to enter a hexadecimal number with four characters at maximum. In next line program print the number at binary form. If user enter out of the range of hexadecimal number; program ask user to enter again.

Example:

TYPE A HEX NUMBER (0000 - FFFF) : xa
ILLEGAL HEX DIGIT, TRY AGAIN ; 1ABC
IN BIRARY IT IS 0001101010111100

4. Write a program ask user to enter a binary number with 16bits at maximum. In next line program print the number at hexadecimal form. If user enters number not binary number (not 0 or 1); program ask user to enter again.

Example:

TYPE A BINARY NUMBER UB TO 16 DIGITS : 112
ILLEGAL BINARY DIGIT , TRY AGAIN : 11100001
IN HEX IT IS : EI

5. Write a program ask user to enter two binary numbers with 8bits at maximum for each one. In next line program print the sum of the two numbers at binary form. If user enters number not binary number (not 0 or 1); program ask user to enter again.

Example:

TYPE A BINARY NUMBER , UP TO 8 DIGITS : 11001010
TYPE A BINARY NUMBER , UP TO 8 DIGITS : 10011100
THE BINARY SUM IS 0000000101100110

6. Write a program ask user to enter decimal numbers until user press enter. In next line program print the sum of the numbers at hexadecimal form. If user enters number not binary number (not 0,1,2,3,…,9); program ask user to enter again.

Example:

ENTER A DECIMAL DIGIT STRING : 1299843
THE SUM OF THE DIGITS IN HEX IS : 0024