ASSEMBLY - 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.

Syntax

Following is the syntax to define a procedure −

```
proc_name:
   procedure body
   ...
   ret
```

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 proc_name
```

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

Example

Let us write a very simple procedure named *sum* that adds the variables stored in the ECX and EDX register and returns the sum in the EAX register −

```
section .text
   global _start        ;must be declared for using gcc

_start:                 ;tell linker entry point
   mov ecx,'4'
   sub     ecx, '0'

   mov  edx, '5'
   sub     edx, '0'

   call    sum           ;call sum procedure
   mov  [res], eax
   mov ecx, msg
   mov edx, len
   mov ebx,1          ;file descriptor (stdout)
   mov eax,4          ;system call number (sys_write)
   int 0x80          ;call kernel

   mov ecx, res
   mov edx, 1
   mov ebx, 1          ;file descriptor (stdout)
   mov eax, 4          ;system call number (sys_write)
   int 0x80          ;call kernel

   mov eax,1          ;system call number (sys_exit)
   int 0x80          ;call kernel
sum:
   mov     eax, ecx
   add     eax, edx
   add     eax, '0'
   ret

section .data
msg db "The sum is:", 0xA,0xD
len equ $- msg
```

```
segment .bss
res resb 1
```

When the above code is compiled and executed, it produces the following result −

```
The sum is:
9
```

Stacks Data Structure

A stack is an array-like data structure in the memory in which data can be stored and removed from a location called the 'top' of the stack. The data that needs to be stored is 'pushed' into the stack and data to be retrieved is 'popped' out from the stack. Stack is a LIFO data structure, i.e., the data stored first is retrieved last.

Assembly language provides two instructions for stack operations: PUSH and POP. These instructions have syntaxes like −

```
PUSH    operand
POP     address/register
```

The memory space reserved in the stack segment is used for implementing stack. The registers SS and ESP *orSP* are used for implementing the stack. The top of the stack, which points to the last data item inserted into the stack is pointed to by the SS:ESP register, where the SS register points to the beginning of the stack segment and the SP *orESP* gives the offset into the stack segment.

The stack implementation has the following characteristics −

- Only **words** or **doublewords** could be saved into the stack, not a byte.
- The stack grows in the reverse direction, i.e., toward the lower memory address
- The top of the stack points to the last item inserted in the stack; it points to the lower byte of the last word inserted.

As we discussed about storing the values of the registers in the stack before using them for some use; it can be done in following way −

```
; Save the AX and BX registers in the stack
PUSH    AX
PUSH    BX

; Use the registers for other purpose
MOV AX, VALUE1
MOV  BX, VALUE2
...
MOV  VALUE1, AX
MOV VALUE2, BX

; Restore the original values
POP AX
POP BX
```

Example

The following program displays the entire ASCII character set. The main program calls a procedure named *display*, which displays the ASCII character set.

```
section .text
   global _start        ;must be declared for using gcc

_start:                 ;tell linker entry point
   call    display
   mov eax,1          ;system call number (sys_exit)
```

```
   int 0x80          ;call kernel

display:
   mov    ecx, 256

next:
   push    ecx
   mov     eax, 4
   mov     ebx, 1
   mov     ecx, achar
   mov     edx, 1
   int     80h

   pop     ecx
   mov dx, [achar]
   cmp byte [achar], 0dh
   inc byte [achar]
   loop    next
   ret

section .data
achar db '0'
```

When the above code is compiled and executed, it produces the following result −

```
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
...
```