Module 5

Assembly Procedures

Subtopic 1

Assembly Procedures

Objectives

- Understand Procedures and Recursion
- Understand Macros in assembly programming
- Apply the concept of File and Memory Management

Assembly — Procedures

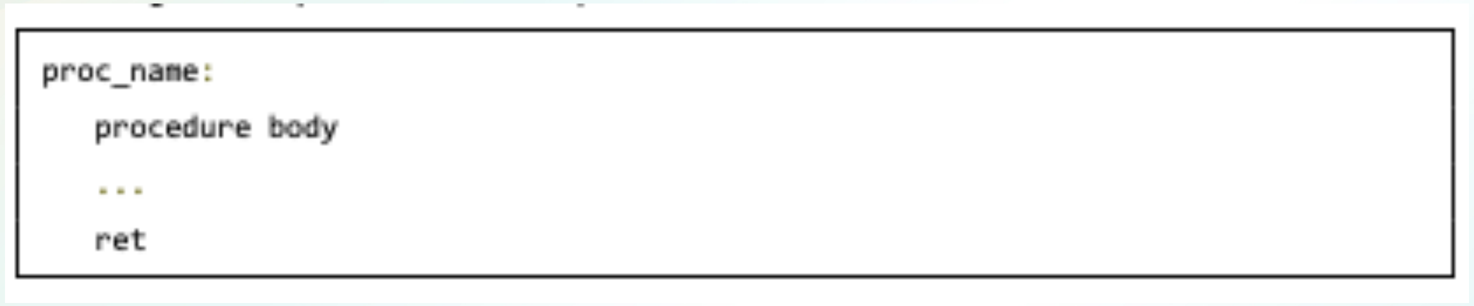
Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size. Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well- defined job. End of the procedure is indicated by a return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

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.

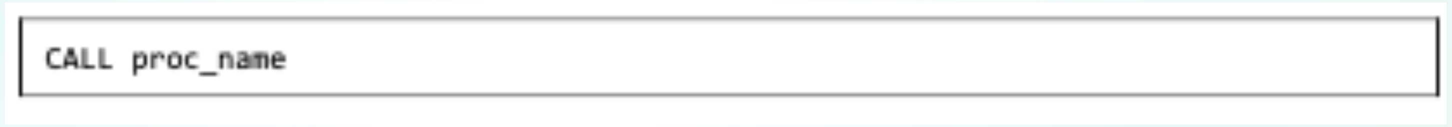
Assembly — Procedures

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:

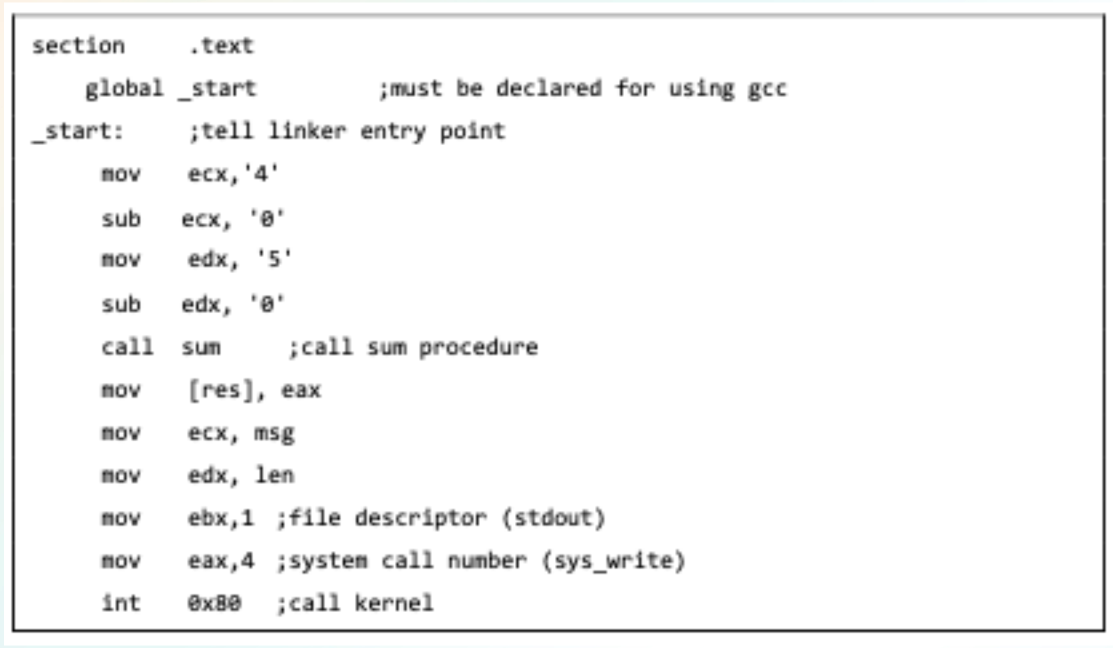


```
CALL proc_name
```

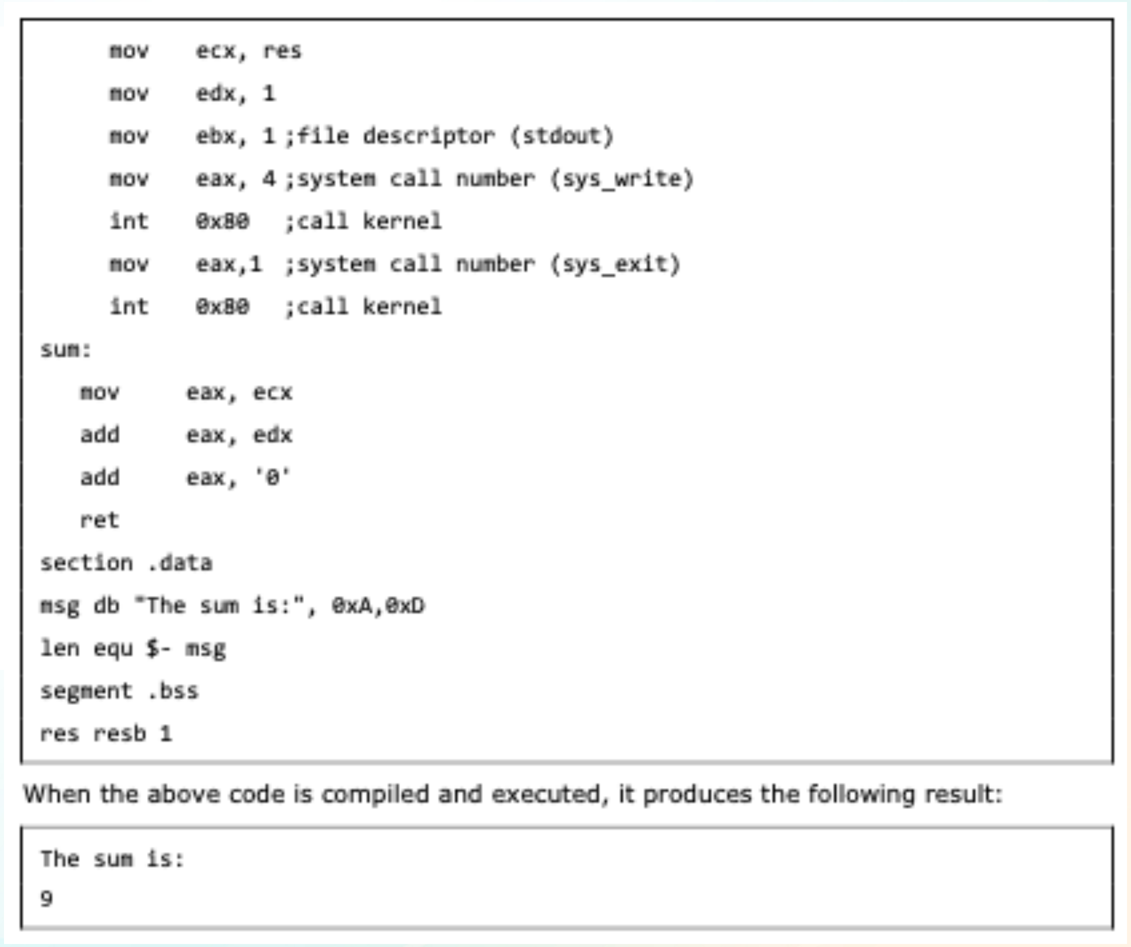
The called procedure returns the control to the calling procedure by using the RET instruction.

Assembly — Procedures

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
```

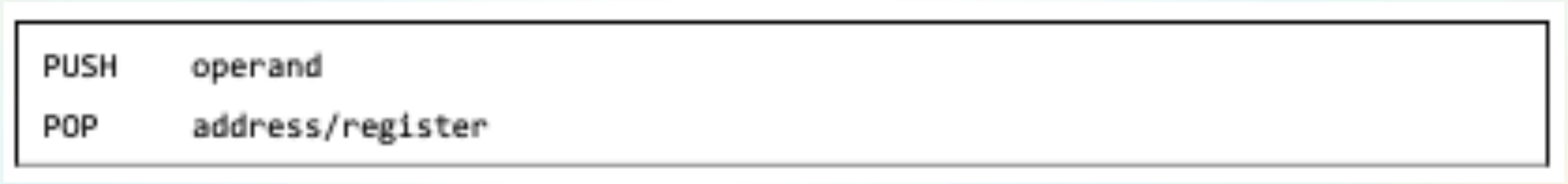
FEU ALABANG FEU DILIMAN FEU TECH

Technology Driven by Innovation

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
```

Stacks Data Structure

The memory space reserved in the stack segment is used for implementing stack. The registers SS and ESP (or SP) are used for implementing the stack.

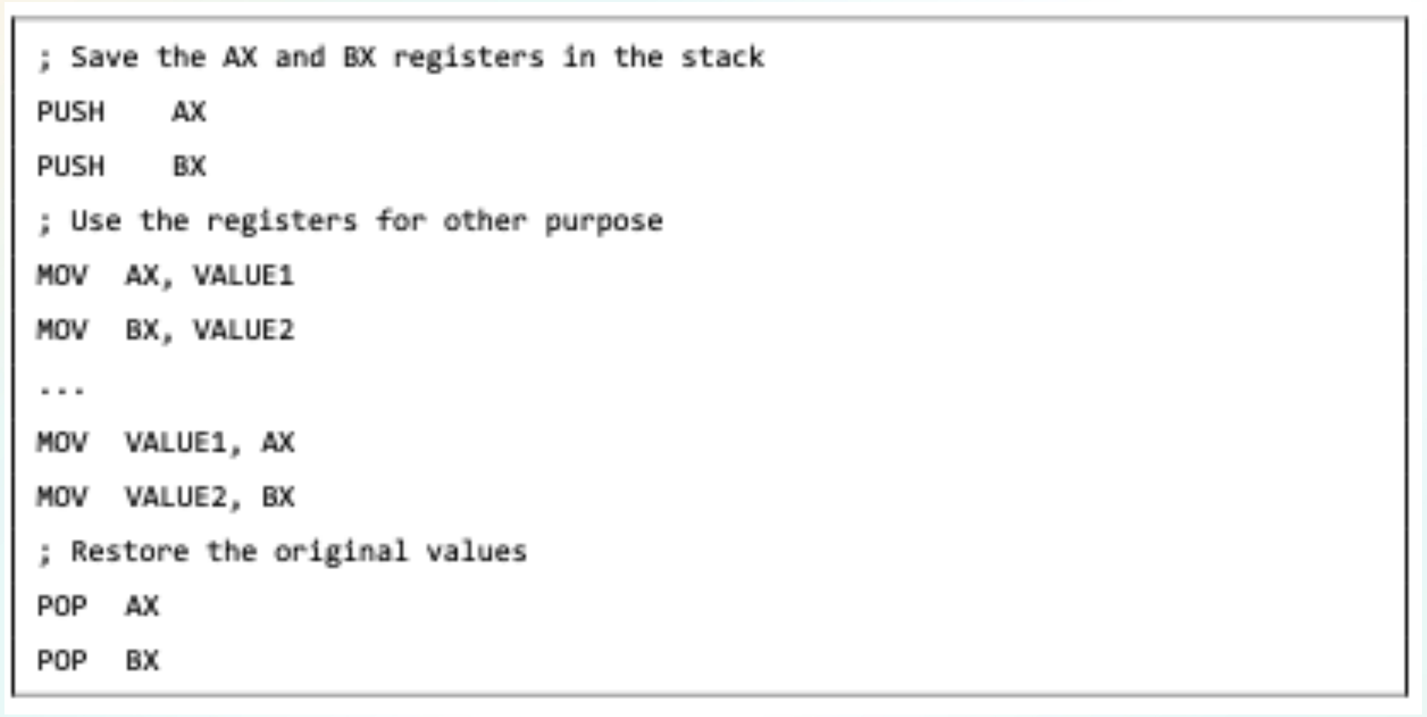
The top of the stack, which points to the last data item inserted into the stack is pointed to by the SS:ESP register, where the SS register points to the beginning of the stack segment and the SP (or ESP) gives the offset into the stack segment.

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.

Stacks Data Structure

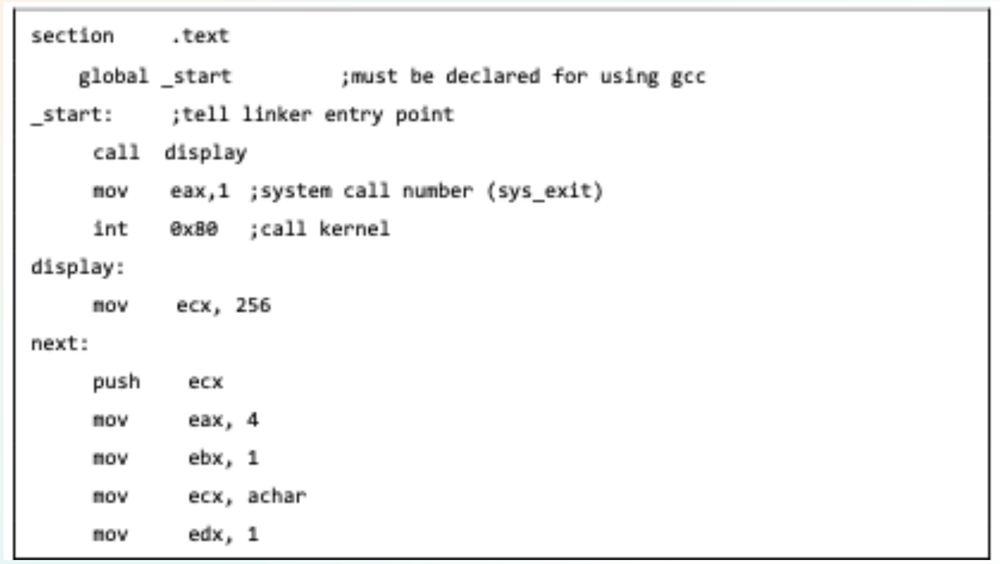
As we discussed about storing the values of the registers in the stack before using them for some use; it can be done in following way:



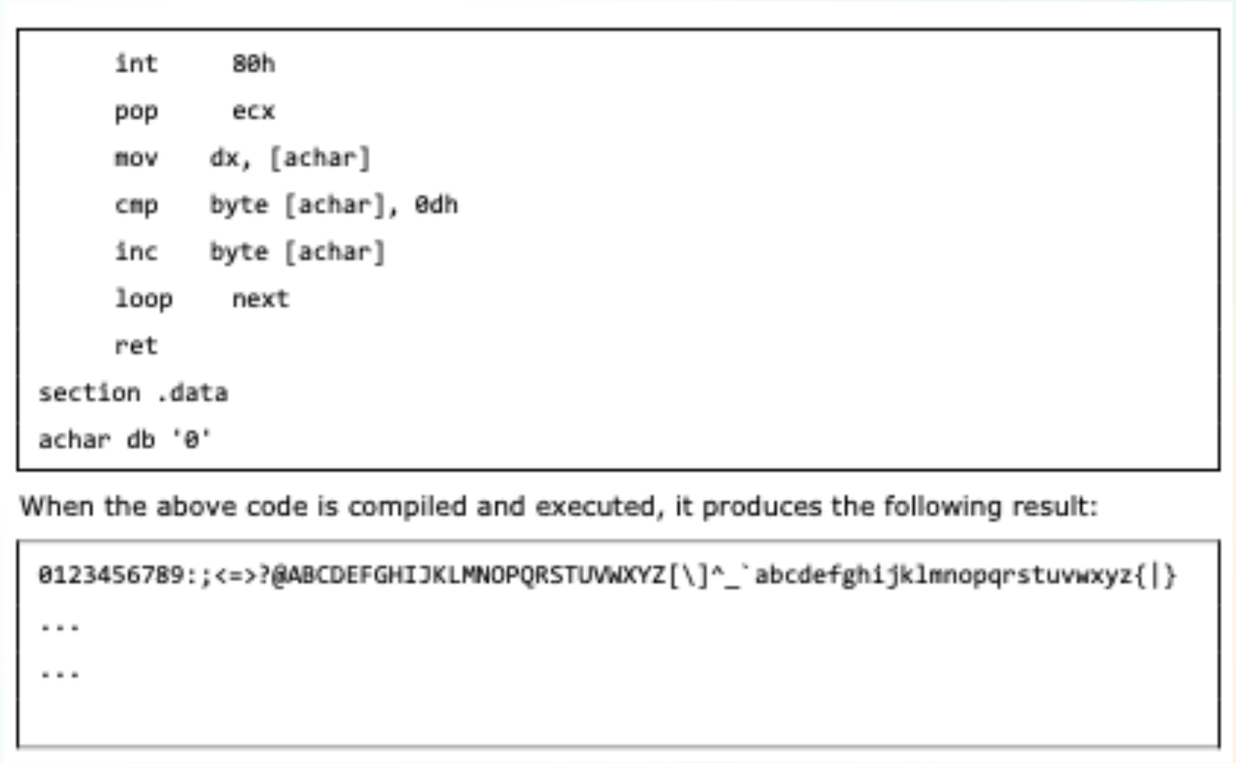
```
; Save the AX and BX registers in the stack
PUSH    AX
PUSH    BX
; Use the registers for other purpose
MOV   AX, VALUE1
MOV   BX, VALUE2
...
MOV   VALUE1, AX
MOV   VALUE2, BX
; Restore the original values
POP   AX
POP   BX
```

Stacks Data Structure

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{|}
...
...
```

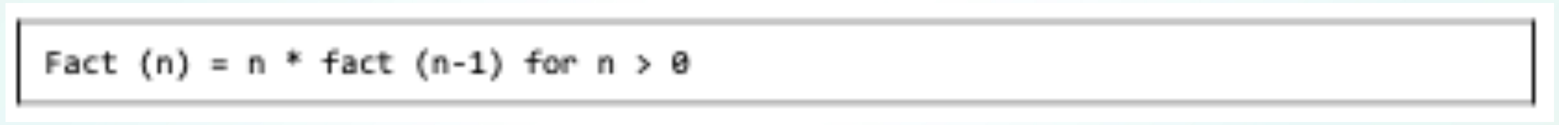
FEU ALABANG FEU DILIMAN FEU TECH

Technology Driven by Innovation

Assembly — Recursion

A recursive procedure is one that calls itself. There are two kind of recursion: direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn calls the first procedure.

Recursion could be observed in numerous mathematical algorithms. For example, consider the case of calculating the factorial of a number. Factorial of a number is given by the equation:

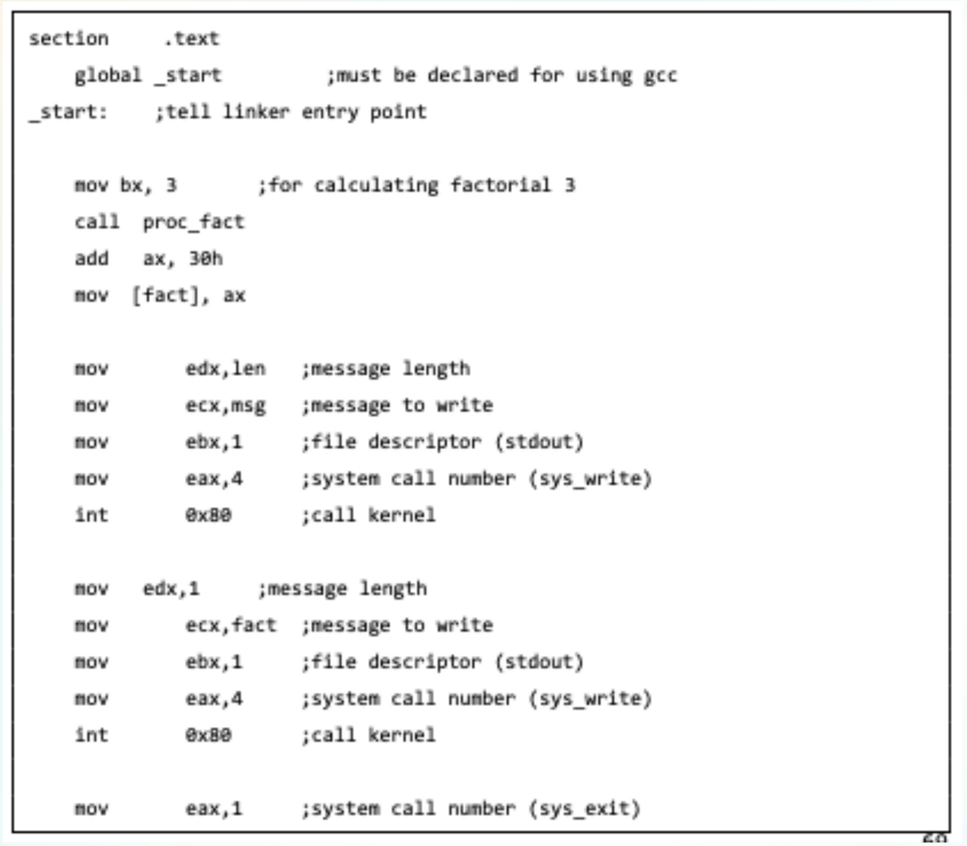


```
Fact (n) = n * fact (n-1) for n > 0
```

For example: factorial of 5 is 1 x 2 x 3 x 4 x 5 = 5 x factorial of 4 and this can be a good example of showing a recursive procedure. Every recursive algorithm must have an ending condition, i.e., the recursive calling of the program should be stopped when a condition is fulfilled. In the case of factorial algorithm, the end condition is reached when n is 0.

Assembly — Recursion

The following program shows how factorial n is implemented in assembly language. To keep the program simple, we will calculate factorial 3.



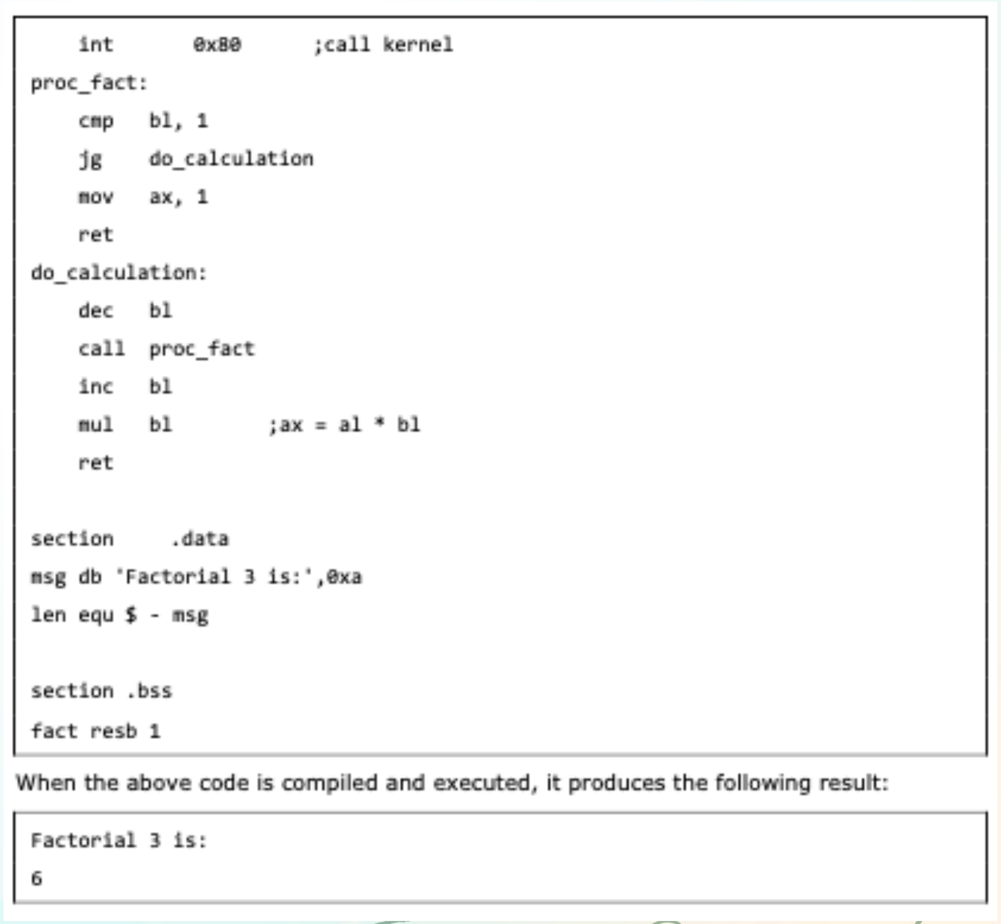
```
section     .text
    global _start         ;must be declared for using gcc
_start:    ;tell linker entry point

    mov bx, 3       ;for calculating factorial 3
    call  proc_fact
    add   ax, 30h
    mov  [fact], ax

    mov       edx,len    ;message length
    mov       ecx,msg    ;message to write
    mov       ebx,1      ;file descriptor (stdout)
    mov       eax,4      ;system call number (sys_write)
    int       0x80       ;call kernel

    mov   edx,1      ;message length
    mov       ecx,fact   ;message to write
    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
proc_fact:
    cmp   bl, 1
    jg    do_calculation
    mov   ax, 1
    ret
do_calculation:
    dec   bl
    call  proc_fact
    inc   bl
    mul   bl          ;ax = al * bl
    ret

section     .data
msg db 'Factorial 3 is:',0xa
len equ $ - msg

section .bss
fact resb 1
```

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

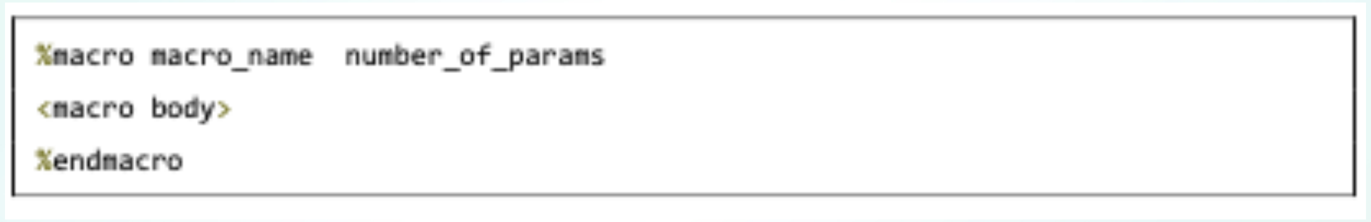
```
Factorial 3 is:
6
```

Assembly — Macros

Writing a macro is another way of ensuring modular programming in assembly language.

- A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.
- In NASM, macros are defined with %macro and %endmacro directives.
- The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition:



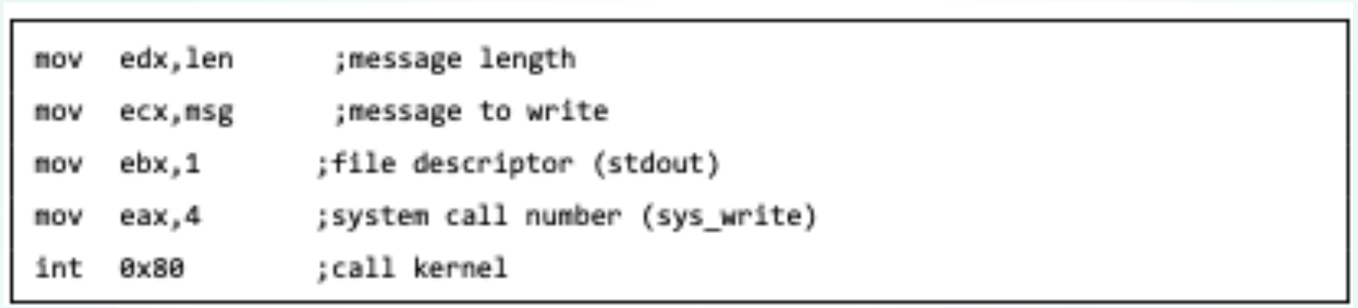
```
%macro macro_name  number_of_params
<macro body>
%endmacro
```

Where, number_of_params specifies the number parameters, macro_name specifies the name of the macro.

Assembly — Macros

The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

For example, a very common need for programs is to write a string of characters in the screen. For displaying a string of characters, you need the following sequence of instructions:



```
mov   edx,len       ;message length
mov   ecx,msg       ;message to write
mov   ebx,1        ;file descriptor (stdout)
mov   eax,4        ;system call number (sys_write)
int   0x80         ;call kernel
```

n the above example of displaying a character string, the registers EAX, EBX, ECX and EDX have been used by the INT 80H function call. So, each time you need to display on screen, you need to save these registers on the stack, invoke INT 80H and then restore the original value of the registers from the stack. So, it could be useful to write two macros for saving and restoring data.

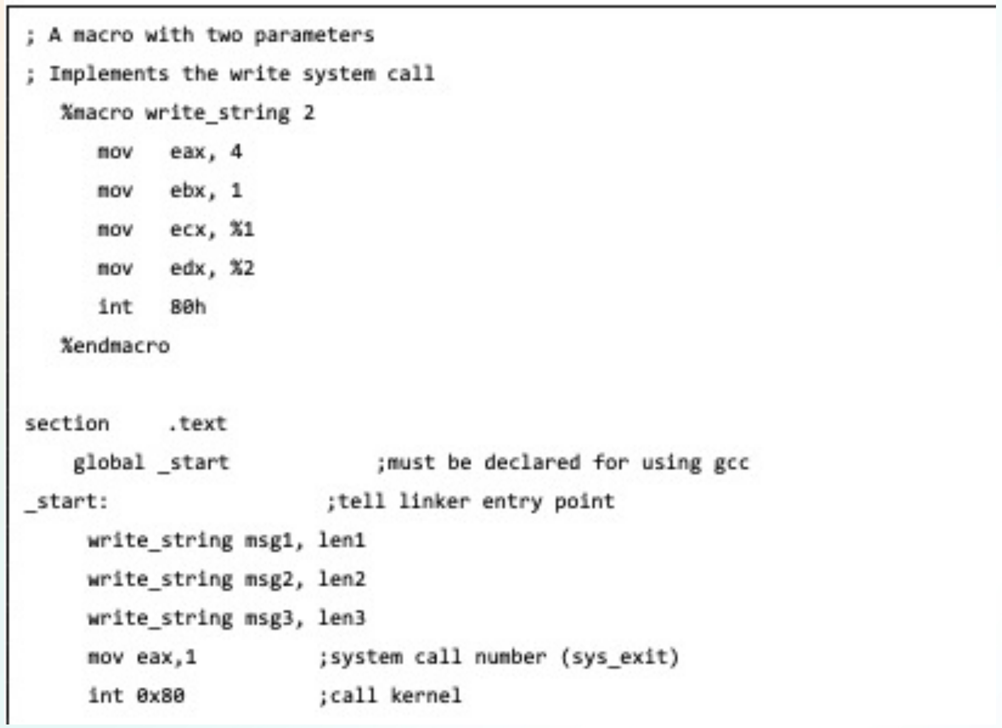
Assembly — Macros

In the given example of displaying a character string, the registers EAX, EBX, ECX and EDX have been used by the INT 80H function call. So, each time you need to display on screen, you need to save these registers on the stack, invoke INT 80H and then restore the original value of the registers from the stack. So, it could be useful to write two macros for saving and restoring data.

We have observed that, some instructions like IMUL, IDIV, INT, etc., need some of the information to be stored in some particular registers and even return values in some specific register(s). If the program was already using those registers for keeping important data, then the existing data from these registers should be saved in the stack and restored after the instruction is executed.

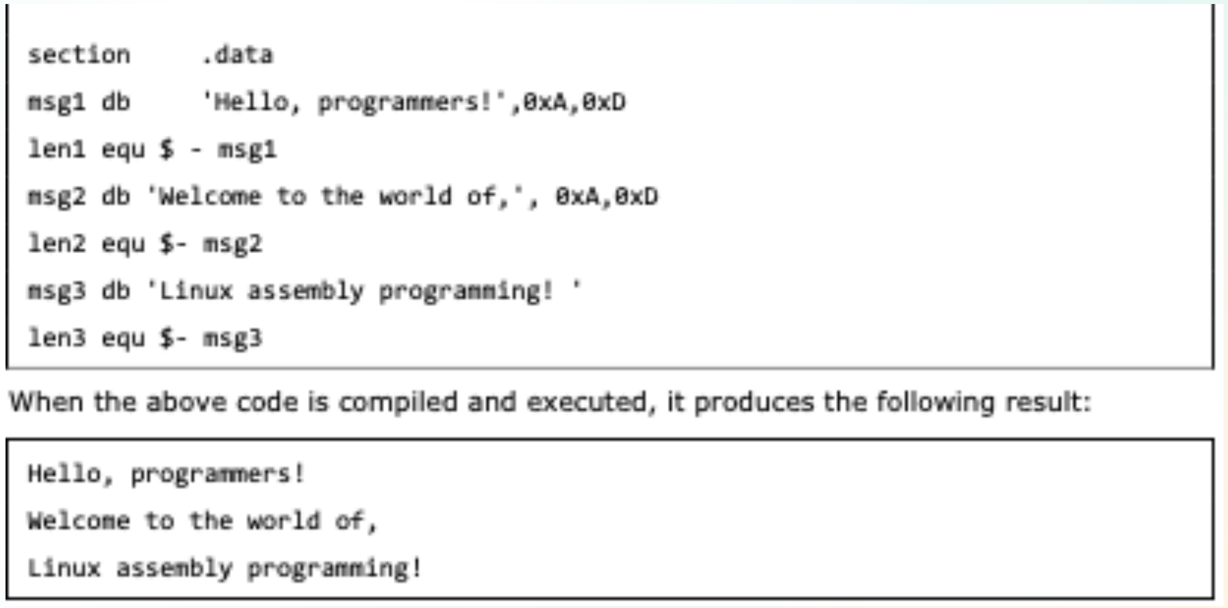
Assembly — Macros

Following example shows defining and using macros:



```
; A macro with two parameters
; Implements the write system call
   %macro write_string 2
      mov   eax, 4
      mov   ebx, 1
      mov   ecx, %1
      mov   edx, %2
      int   80h
   %endmacro

section     .text
    global _start             ;must be declared for using gcc
_start:                      ;tell linker entry point
     write_string msg1, len1
     write_string msg2, len2
     write_string msg3, len3
     mov eax,1               ;system call number (sys_exit)
     int 0x80                ;call kernel
```



```
section     .data
msg1 db     'Hello, programmers!',0xA,0xD
len1 equ $ - msg1
msg2 db 'Welcome to the world of,', 0xA,0xD
len2 equ $- msg2
msg3 db 'Linux assembly programming! '
len3 equ $- msg3
```

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

```
Hello, programmers!
Welcome to the world of,
Linux assembly programming!
```

FEU ALABANG FEU DILIMAN FEU TECH

Technology Driven by Innovation

Assembly — File Management

The system considers any input or output data as stream of bytes.

There are three standard file streams:

- Standard input (stdin),
- Standard output (stdout), and
- Standard error (stderr).

Assembly — File Management

File Descriptor

A file descriptor is a 16-bit integer assigned to a file as a file id. When a new file is created or an existing file is opened, the file descriptor is used for accessing the file.

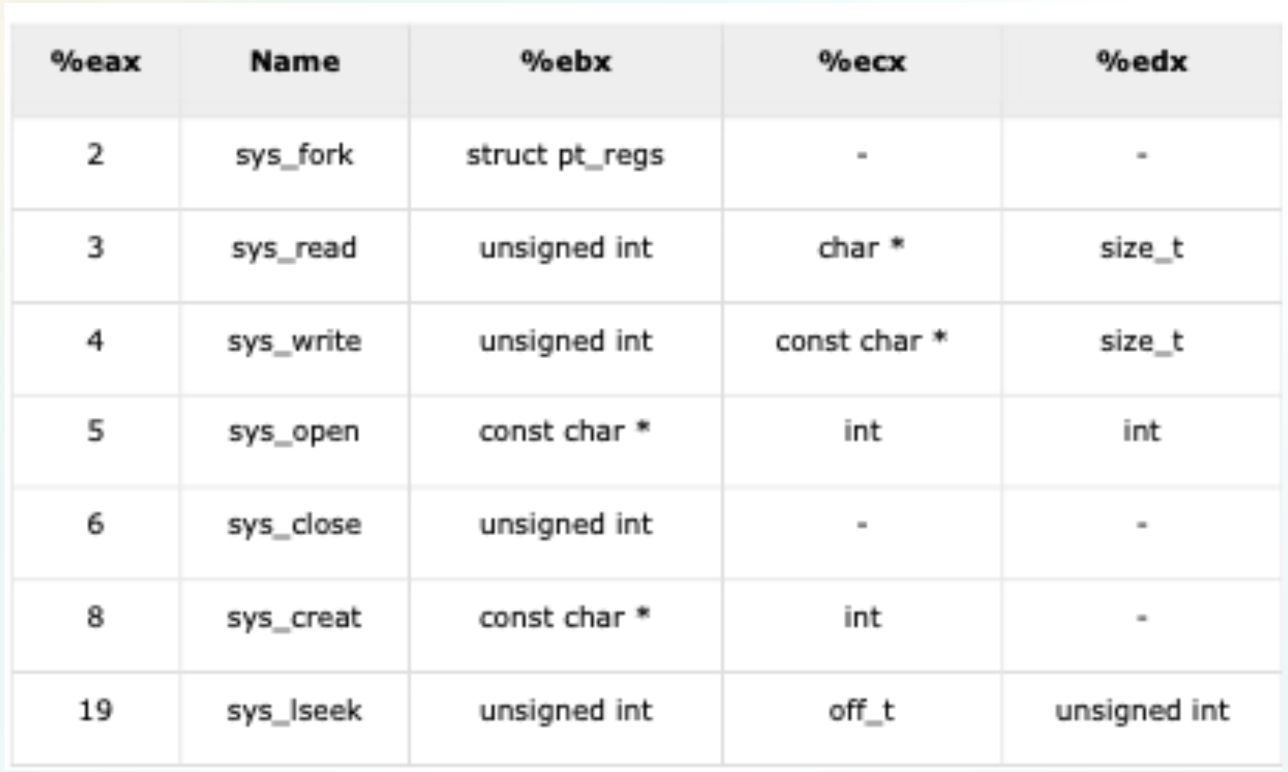
File descriptor of the standard file streams - stdin, stdout and stderr are 0, 1 and 2, respectively.

File Pointer

A file pointer specifies the location for a subsequent read/write operation in the file in terms of bytes. Each file is considered as a sequence of bytes. Each open file is associated with a file pointer that specifies an offset in bytes, relative to the beginning of the file. When a file is opened, the file pointer is set to zero.

File Handling System Calls

The following table briefly describes the system calls related to file handling:



%eax	Name	%ebx	%ecx	%edx
2	sys_fork	struct pt_regs	-	-
3	sys_read	unsigned int	char *	size_t
4	sys_write	unsigned int	const char *	size_t
5	sys_open	const char *	int	int
6	sys_close	unsigned int	-	-
8	sys_creat	const char *	int	-
19	sys_lseek	unsigned int	off_t	unsigned int

File Handling System Calls

The steps required for using the system calls are same, as we discussed earlier: 1.

1. Put the system call number in the EAX register.
2.StoretheargumentstothesystemcallintheregistersEBX,ECX, etc.
3. Call the relevant interrupt(80h).
4. The result is usually returned in the EAX register.

File Handling System Calls

Creating and Opening a File

For creating and opening a file, perform the following tasks:

1. Putthesystemcallsys_creat()number8,intheEAXregister

2. PutthefilenameintheEBXregister

3. PutthefilepermissionsintheECXregister

The system call returns the file descriptor of the created file in the EAX register, in case of error, the error code is in the EAX register.

File Handling System Calls

Opening an Existing File

For opening an existing file, perform the following tasks:

1. Put the system callsys_open() number 5, in the EAX register.

2. Put the filename in the EBX register.

3. Put the file access mode in the ECX register.

4. Put the file permissions in the EDX register.

The system call returns the file descriptor of the created file in the EAX register, in case of error, the error code is in the EAX register.
Among the file access modes, most commonly used are: read-only (0), write-only (1), and read-write (2).

File Handling System Calls

Reading from a File

For reading from a file, perform the following tasks:

1. Put the system call sys_read() number 3, in the EAX register.

2. Put the file descriptor in the EBX register.

3. Put the pointer to the input buffer in the ECX register.

4. Put the buffer size, i.e., the number of bytes to read, in the EDX register.

The system call returns the number of bytes read in the EAX register, in case of error, the error code is in the EAX register.

File Handling System Calls

Writing to a File

For writing to a file, perform the following tasks:

1. Put the system call sys_write() number 4, in the EAX register.

2. Put the file descriptor in the EBX register.

3. Put the pointer to the output buffer in the ECX register.

4. Put the buffersize,i.e., the number of bytes to write, in the EDX register.

The system call returns the actual number of bytes written in the EAX register, in case of error, the error code is in the EAX register.

File Handling System Calls

Closing a File

For closing a file, perform the following tasks:

1. Put the system callsys_close() number 6, in the EAX register.

2. Put the file descriptor in the EBX register.

The system call returns, in case of error, the error code in the EAX register.

File Handling System Calls

Updating a File

For updating a file, perform the following tasks:

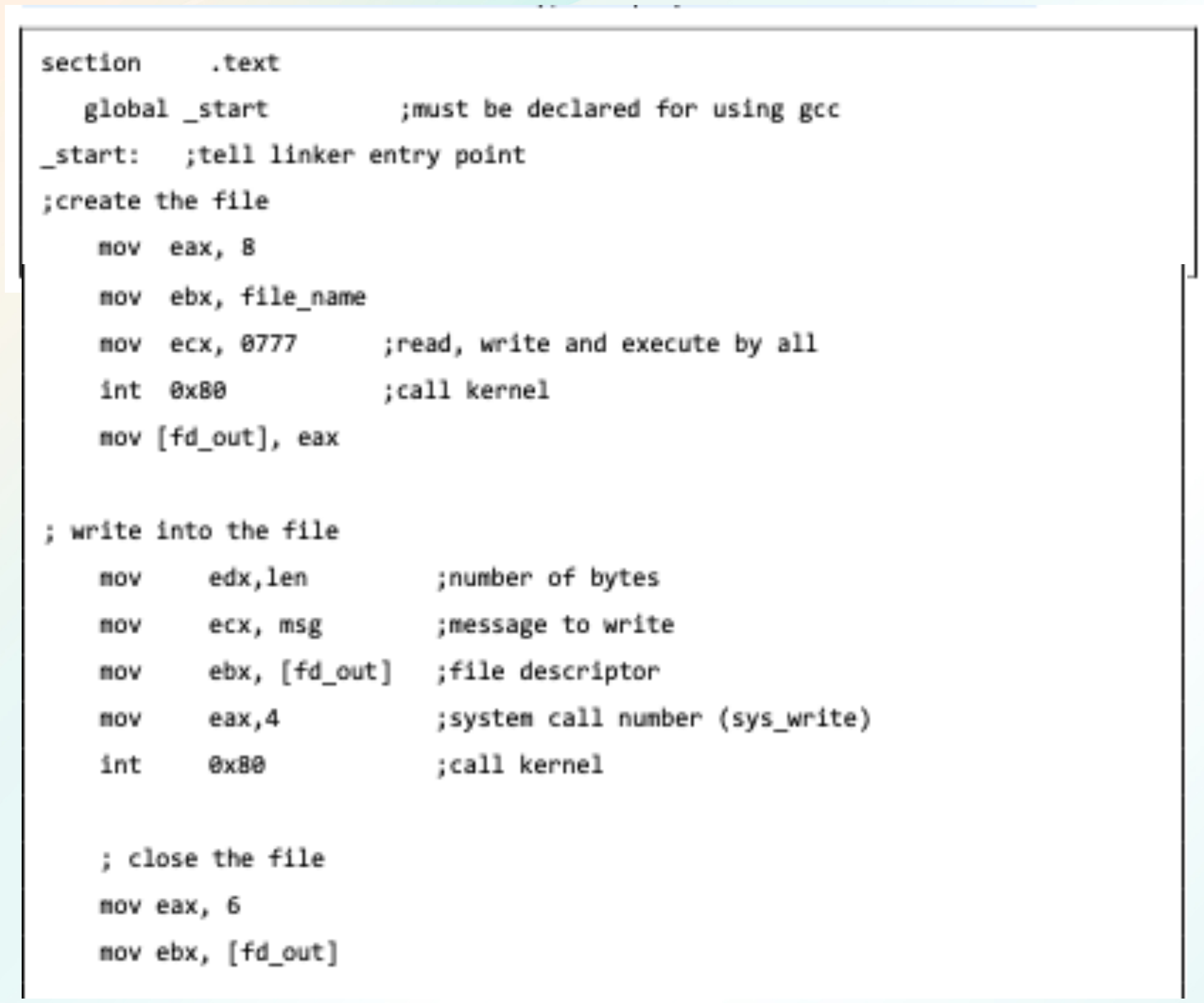
1. Put the system call sys_lseek() number 19, in the EAX register.
2. Put the file descriptor in the EBX register.
3. Put the offset value in the ECX register.
4. Put the reference position for the offset in the EDX register.

The reference position could be:

- Beginning of file - value 0
- Current position - value 1
- End of file - value 2

The system call returns, in case of error, the error code in the EAX register.

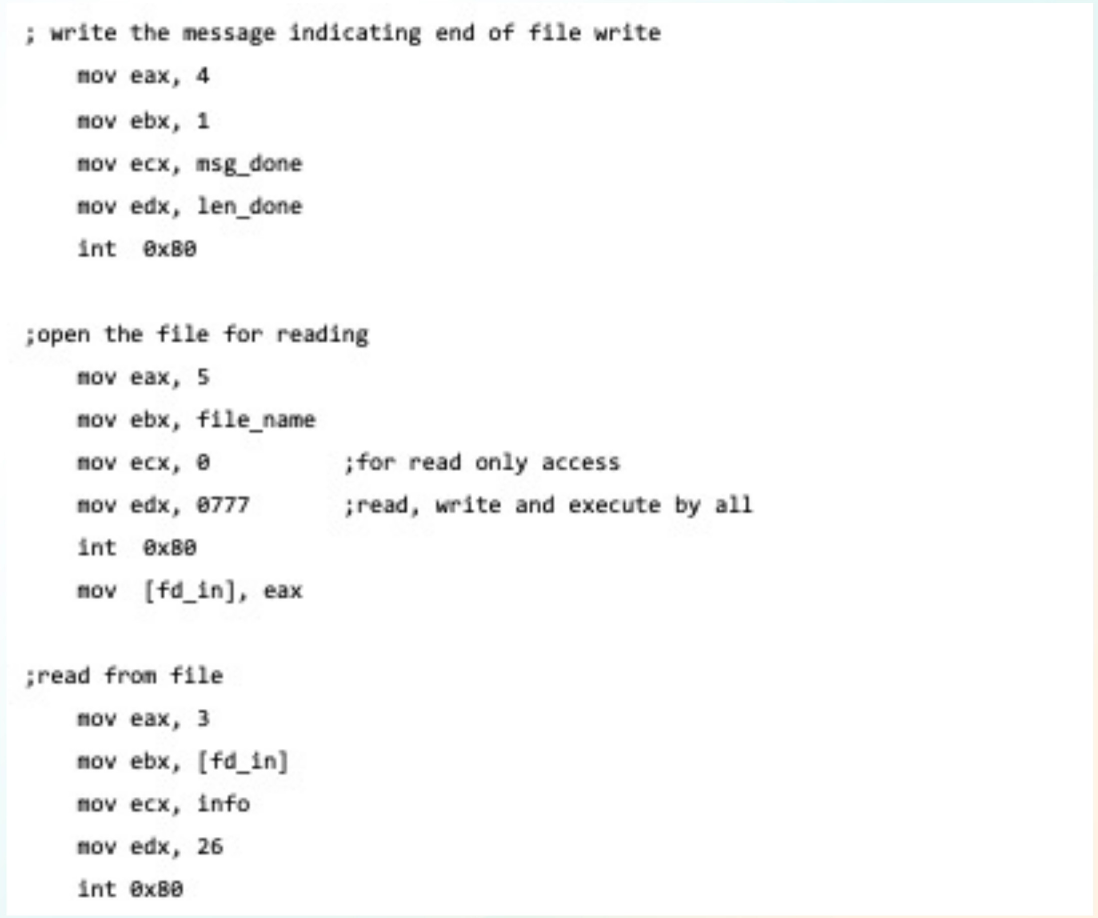
The following program creates and opens a file named myfile.txt, and writes a text 'Welcome to Tutorials Point' in this file. Next, the program reads from the file and stores the data into a buffer named info. Lastly, it displays the text as stored in info.



```
section    .text
   global _start         ;must be declared for using gcc
_start:   ;tell linker entry point
;create the file
    mov  eax, 8
    mov  ebx, file_name
    mov  ecx, 0777      ;read, write and execute by all
    int  0x80           ;call kernel
    mov [fd_out], eax

; write into the file
    mov     edx,len          ;number of bytes
    mov     ecx, msg         ;message to write
    mov     ebx, [fd_out]    ;file descriptor
    mov     eax,4            ;system call number (sys_write)
    int     0x80             ;call kernel

    ; close the file
    mov eax, 6
    mov ebx, [fd_out]
```

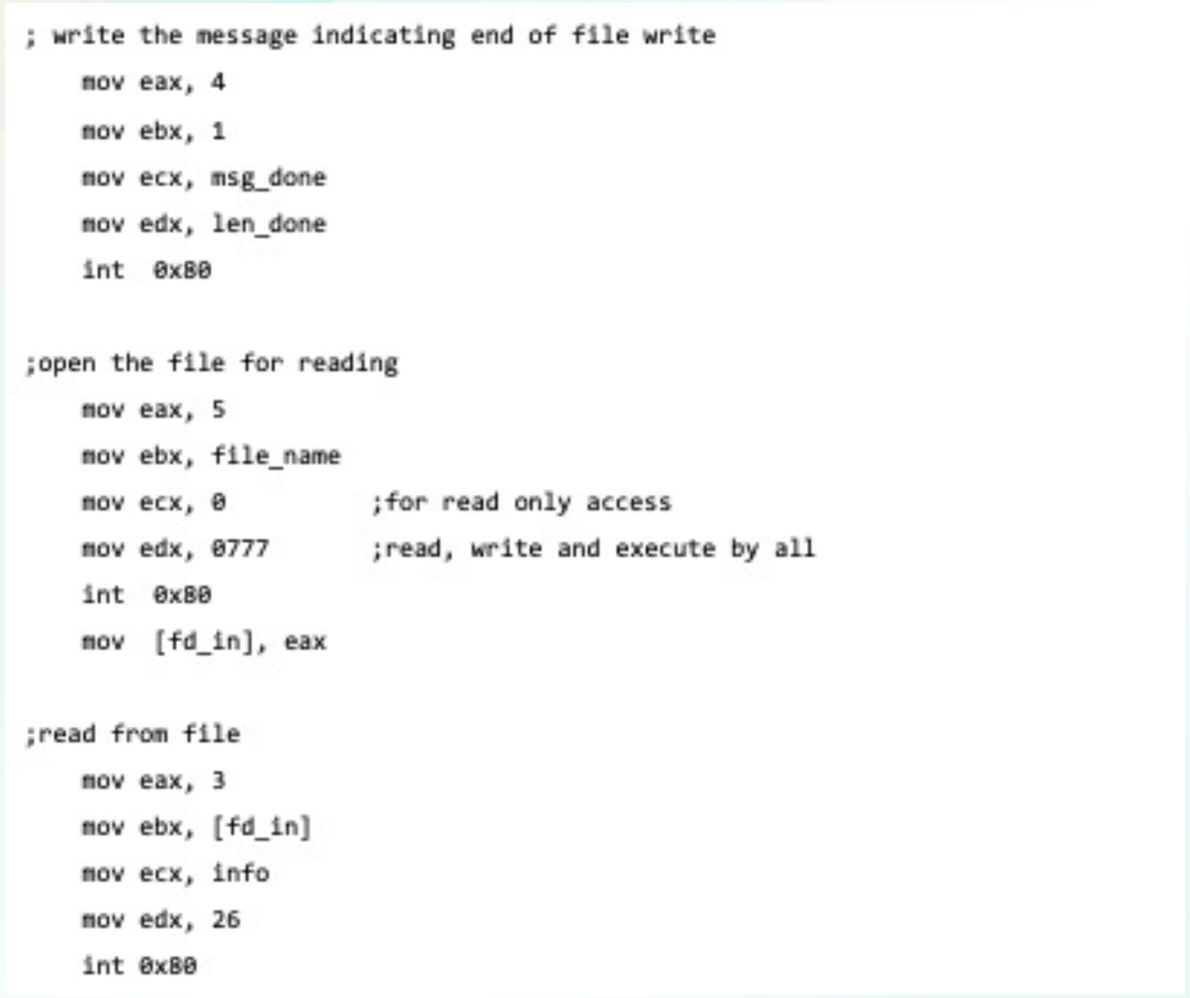


```
; write the message indicating end of file write
    mov eax, 4
    mov ebx, 1
    mov ecx, msg_done
    mov edx, len_done
    int  0x80

;open the file for reading
    mov eax, 5
    mov ebx, file_name
    mov ecx, 0          ;for read only access
    mov edx, 0777       ;read, write and execute by all
    int  0x80
    mov  [fd_in], eax

;read from file
    mov eax, 3
    mov ebx, [fd_in]
    mov ecx, info
    mov edx, 26
    int 0x80
```

The following program creates and opens a file named myfile.txt, and writes a text 'Welcome to Tutorials Point' in this file. Next, the program reads from the file and stores the data into a buffer named info. Lastly, it displays the text as stored in info.

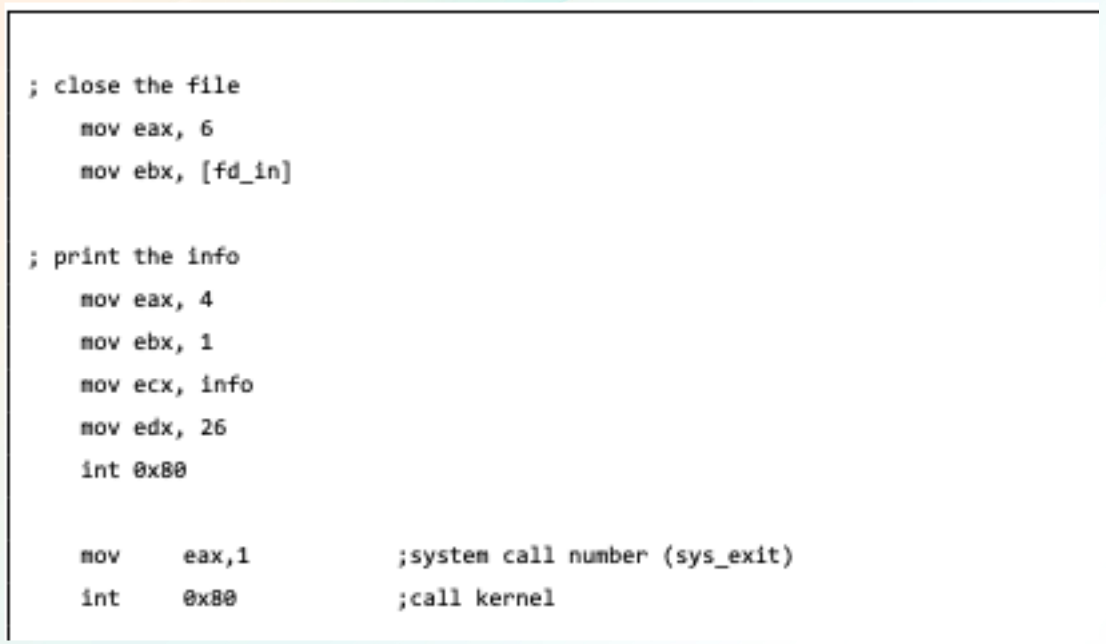


```
; write the message indicating end of file write
    mov eax, 4
    mov ebx, 1
    mov ecx, msg_done
    mov edx, len_done
    int  0x80

;open the file for reading
    mov eax, 5
    mov ebx, file_name
    mov ecx, 0          ;for read only access
    mov edx, 0777       ;read, write and execute by all
    int  0x80
    mov  [fd_in], eax

;read from file
    mov eax, 3
    mov ebx, [fd_in]
    mov ecx, info
    mov edx, 26
    int 0x80
```

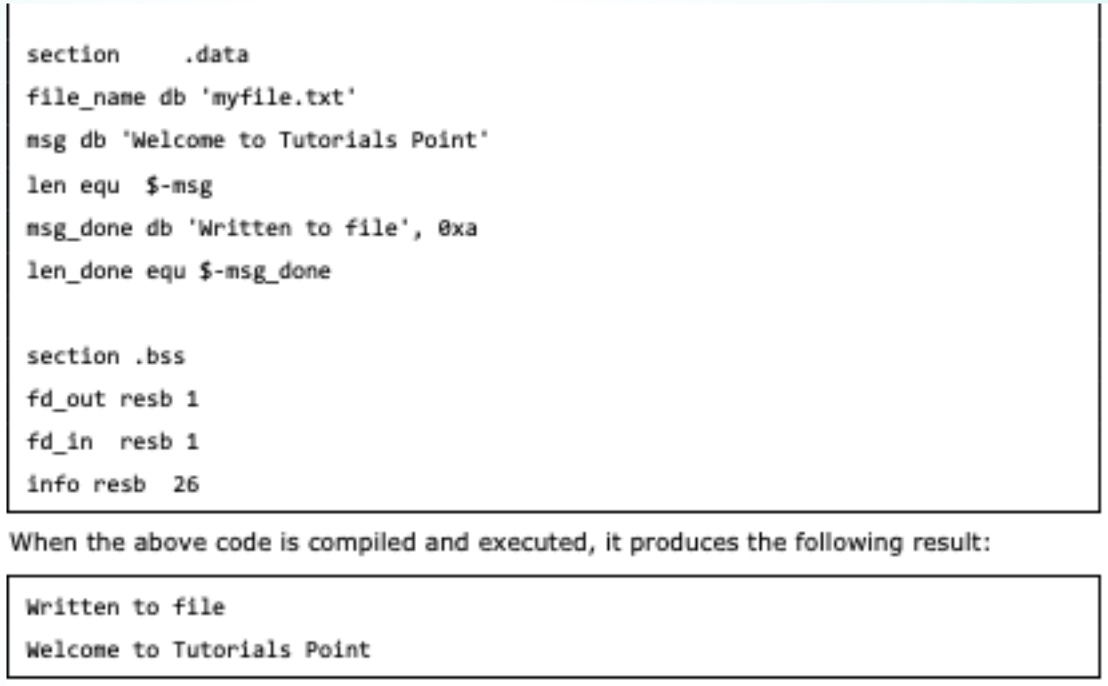
The following program creates and opens a file named myfile.txt, and writes a text 'Welcome to Tutorials Point' in this file. Next, the program reads from the file and stores the data into a buffer named info. Lastly, it displays the text as stored in info.



```
; close the file
    mov eax, 6
    mov ebx, [fd_in]

; print the info
    mov eax, 4
    mov ebx, 1
    mov ecx, info
    mov edx, 26
    int 0x80

    mov     eax,1           ;system call number (sys_exit)
    int     0x80            ;call kernel
```



```
section     .data
file_name db 'myfile.txt'
msg db 'Welcome to Tutorials Point'
len equ  $-msg
msg_done db 'Written to file', 0xa
len_done equ $-msg_done

section .bss
fd_out resb 1
fd_in  resb 1
info resb  26
```

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

```
Written to file
Welcome to Tutorials Point
```

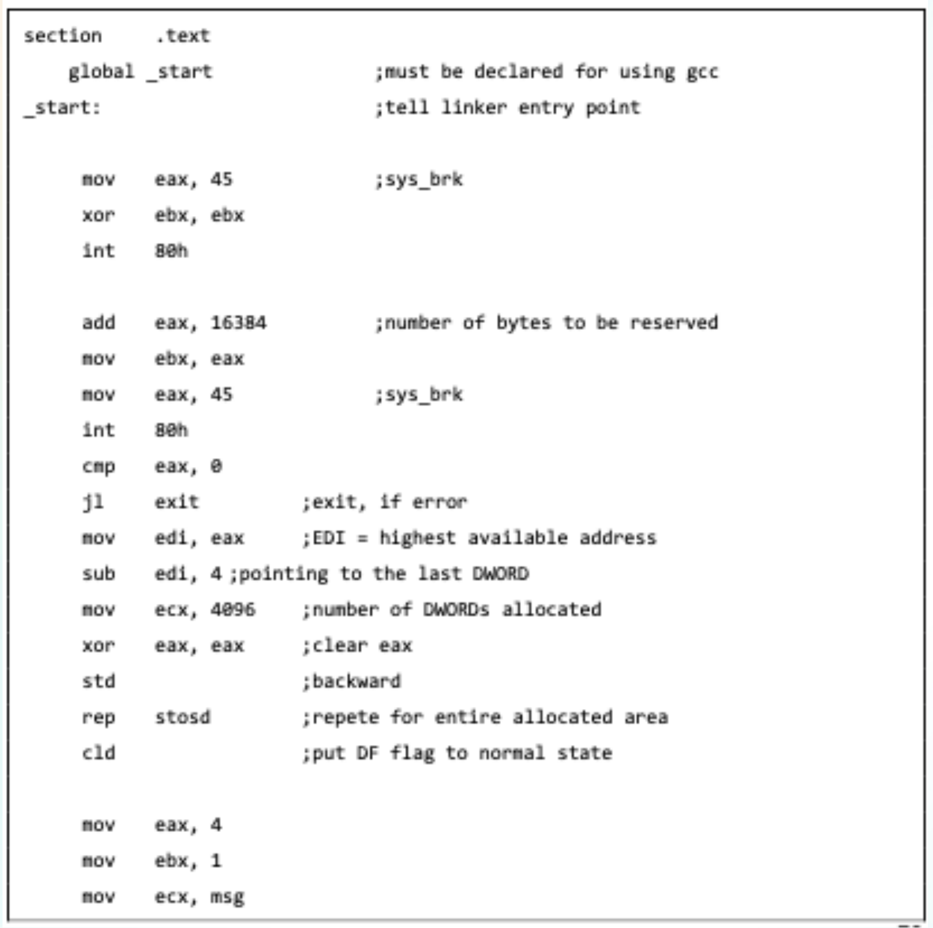
Memory Management

The sys_brk() system call is provided by the kernel, to allocate memory without the need of moving it later. This call allocates memory right behind the application image in the memory. This system function allows you to set the highest available address in the data section.

This system call takes one parameter, which is the highest memory address needed to be set. This value is stored in the EBX register.
In case of any error, sys_brk() returns -1 or returns the negative error code itself. The following example demonstrates dynamic memory allocation.

Memory Management

The following program allocates 16kb of memory using the sys_brk() system call:

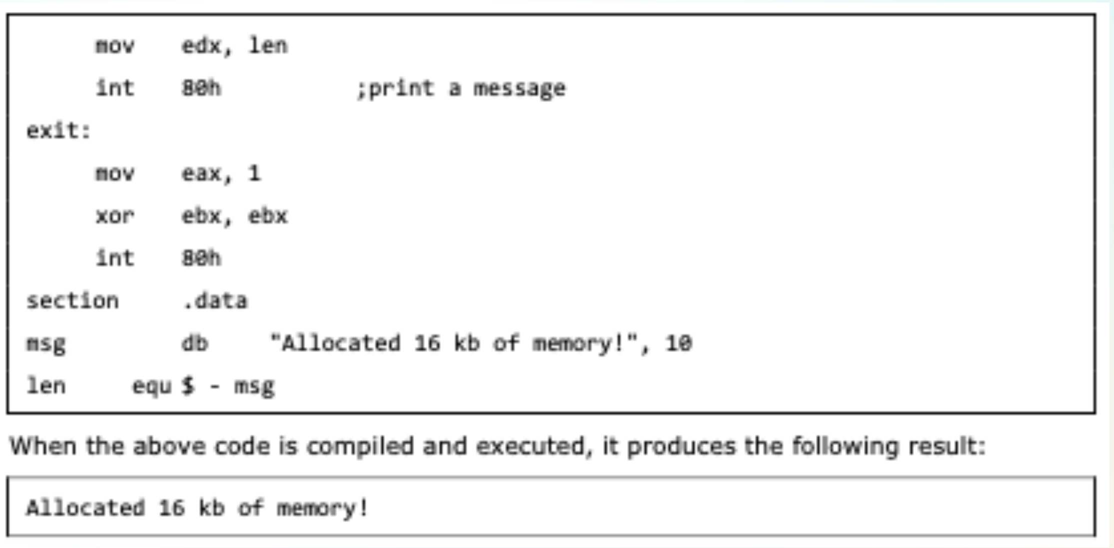


```
section     .text
    global _start               ;must be declared for using gcc
_start:                         ;tell linker entry point

     mov    eax, 45             ;sys_brk
     xor    ebx, ebx
     int    80h

     add    eax, 16384          ;number of bytes to be reserved
     mov    ebx, eax
     mov    eax, 45             ;sys_brk
     int    80h
     cmp    eax, 0
     jl     exit          ;exit, if error
     mov    edi, eax      ;EDI = highest available address
     sub    edi, 4 ;pointing to the last DWORD
     mov    ecx, 4096     ;number of DWORDs allocated
     xor    eax, eax      ;clear eax
     std                  ;backward
     rep    stosd         ;repete for entire allocated area
     cld                  ;put DF flag to normal state

     mov    eax, 4
     mov    ebx, 1
     mov    ecx, msg
```



```
     mov    edx, len
     int    80h         ;print a message
exit:
     mov    eax, 1
     xor    ebx, ebx
     int    80h
section     .data
msg         db    "Allocated 16 kb of memory!", 10
len      equ $ - msg
```

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

```
Allocated 16 kb of memory!
```