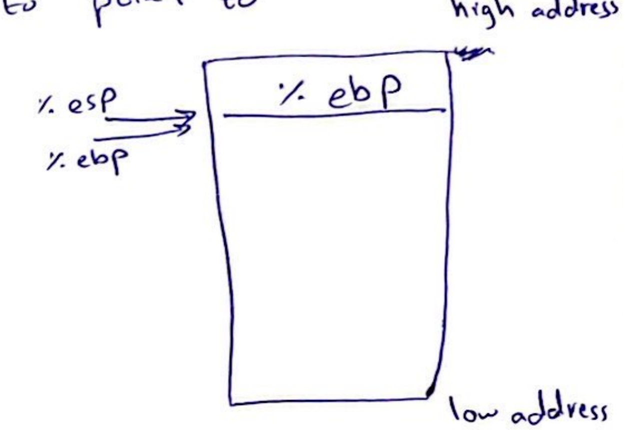
_main :

LFB13 :

```
pushl %ebp
movl %esp, %ebp
```

# pushing the value of the base stack pointer into the stack (decrement %esp by 4)
# and then move the value of %esp to %ebp which sets the base pointer to point to the same location as %esp

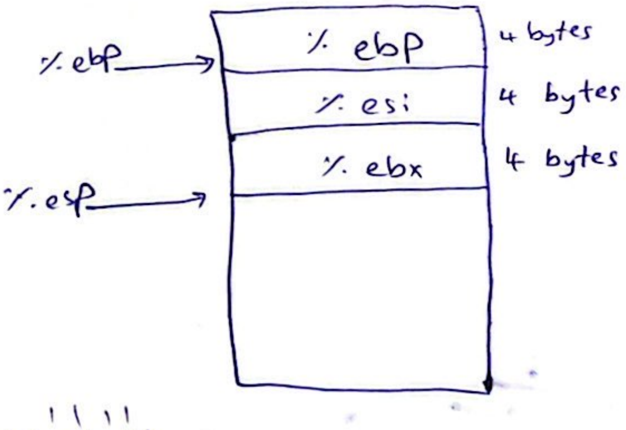


```
pushl %esi
```

# decrement the stack pointer %esp by 4 and push the 32-bit %esi value into the stack

```
pushl %ebx
```

# decrement %esp by 4 and push %ebx value



```
andl $-16, %esp
```

# 16 → $\underbrace{\ldots 0010000}_{32\text{-bits}} \xrightarrow{2's\ comp} \ldots 11101111$

$+ \; 1$

$\underbrace{\ldots 11110000}_{32\text{-bits}}$

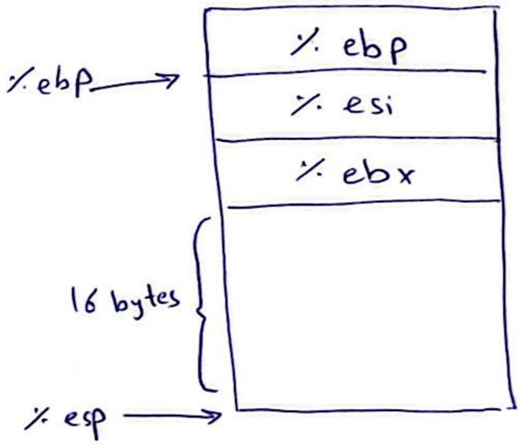
# So -16 in hexa is 0xfffffff0 → %esp = %esp &0xffffff

# when anding the %esp register with 0xfffffff0 we set the %esp register to the nearest multiple of 16 (stack alignment) which is sometimes requried to improve performance for processors that operate more efficietly when data is aligned.

# reference : https://stackoverflow.com/questions/23309863/why-does-gcc-produce-andl-16

```
subl $16, %esp
```

# %esp = %esp-16 , allocate 16 bytes in the stack (for local variables



```
call ___main
```

# this line refers to the ___main in the C library which eventually calls the main() function (internal procedure)

# reference: https://community.st.com/t5/stm32-mcus-products/main-in-startup-assembly/td-p/391658

```
movl $1 , %ebx
movl $0 , %esi
```

# %ebx = 1 , i is now stored in %ebx

# %esi = 0 , result in %esi

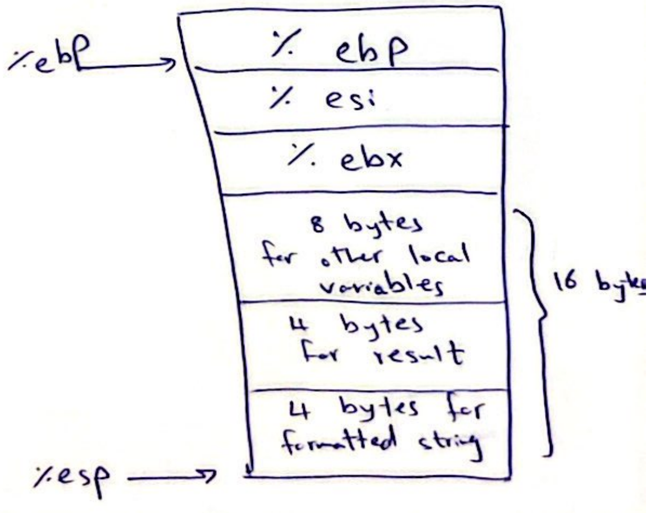
```
jmp L7
```

# jump to L7

L7:

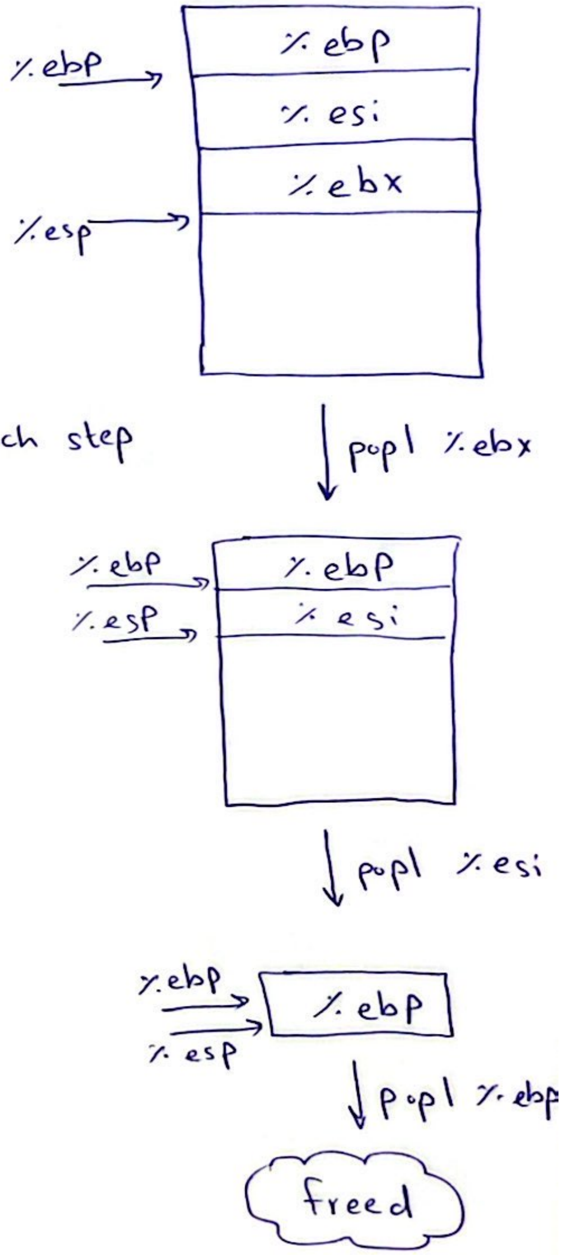
| result | %esi |
|---|---|
| i | %ebx |

```
    cmpl $5, %ebx
    # i : 5
    # compare i with 5
    jle L8
    # if less than or equal then jump to L8 which is the
      loop body (i ≤ 5) ⟶ L8 explained in page ~~[illegible]~~ (5)
    # we noticed that the compiler didn't save the value
      of the variable "number" in any register and dealt with
      it as a constant
    # the compiler optimized the code by treating that variable
      as a compile time constant rather than storing it in a
      register

    movl %esi, 4(%esp)
    # move the content in the register %esi to the
      memory location of (4+%esp), (4+%esp) = %esi
                                   memory
                                   location
    # 4 bytes from the stack pointer, preparing it to
      be passed to printf

    movl $LC0, (%esp)
    # the format string "%d" is stored at the memory location
      pointed to by %esp to be prepared ~~to be pr~~ to be passed
      to printf, the "%d" in the format string suggest that it
      is expecting integar argument
    #$LC0 is explained in details in page (6)

    call _printf
    # to print result
```


%ebp
% ebp
% esi
% ebx
8 bytes for other local variables
4 bytes for result
4 bytes for formatted string
16 bytes
%esp

④

```
movl  $0, %eax

# %eax = 0 , moving the immediate value 0 to %eax
  for returning 0 at the end of the main function


leal -8 (%ebp), %esp

# %esp = %ebp - 8   (cleaning up the stack)

popl %ebx

# increment %esp by 4 and
  storing the value at %ebx

popl %esi

popl %ebp

# same thing incrementing %esp by 4 each step
  to free the stack
```



```
ret

# returning the value in the register %eax
  which is 0   (return 0;)
```

#if i ≤ 5 then we jump to L8

| ~~%~~ i | % ebx |
|---|---|
| result | % esi |

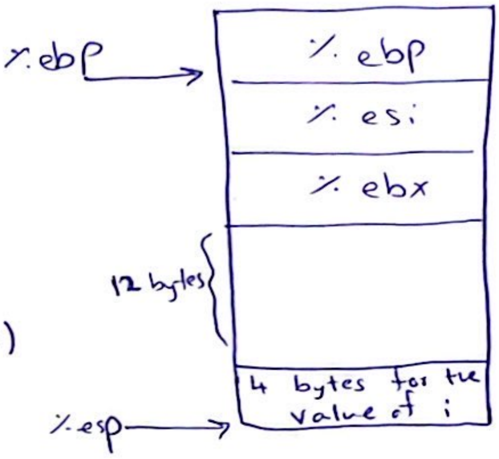
L8 :

movl % ebx , (% esp)

# move the value of % ebx (i) to the memory location of the stack pointer to use it by Factorial function

call _factorial

# calling the function → explained in page ⑦



addl % eax, % esi

# % esi = % esi + % eax → result += factorial (i)

~~# % esi is~~ .

result

the value returned from the function

~~#~~ addl $1, % ebx

# % ebx = % ebx + 1 → i++

# then L8 body is completed it will go again to L7 and does comparison with 5 (see the code)

if i ≤ 5
goto L8

```
L8 :
    movl  % ebx , (% esp)
    call _factorial
    addl % eax, % esi  #result+= factorial(
    addl  $ 1 , % ebx    # i++
L7 :
     cmpl   $ 5, % ebx  # i:5
     jle  L8
     - -
     - -
     -
```

#.I : <5 then we jump to L8

# before we jump to explaing factorial function
# we explained LCO because it is an important part of the program
# this is extra information

.ascii "%d\0"

# include these characters in the data section of the program

.text

# starts a new section, the "text" section is the section in object files that stores code

.global _main

# tells the assembler that this is a global symbol and should be visible to the linker because other object files will use it (declares it as a global symbol)

.def _main ; .scl 2 ; . type 32 ; . endef

# .def defines a symbol _main

# .scl 2 set the storage class of the symbol to 2 (external symbol) external symbol means that is defined or used in another module

# .type 32 indicates that _main is a function symbol (sets the type of the symbol) number 32 indicates a function

# . endef : end definition

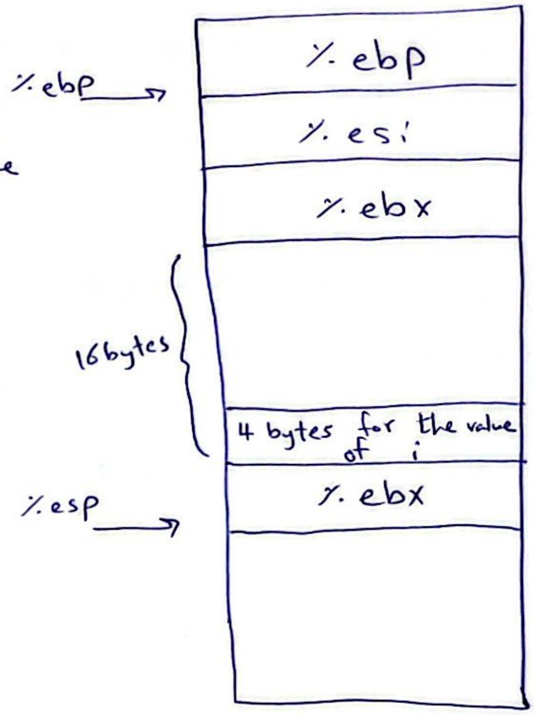
# reference : https://stackoverflow.com/questions/17794533/what-does-this-assembly-language-code-mean

- factorial:

LFB12:

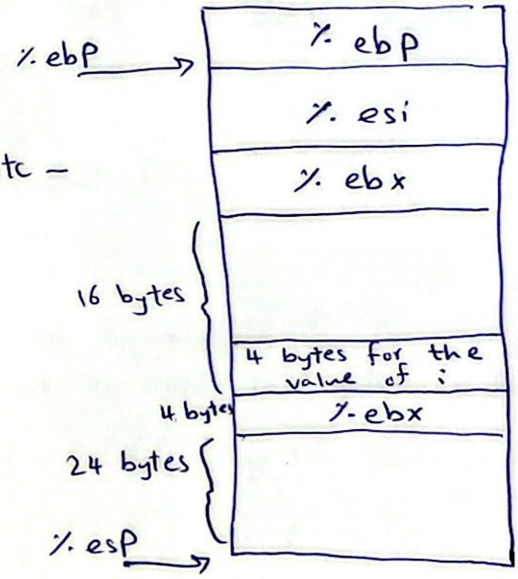
pushl %ebx

# decrement esp by 4 to push %ebx value



subl $24, %esp

# decrement %esp by 24
# %esp = %esp - 24
# allocating 24 bytes for local variables etc -



movl 32 (%esp), %ebx

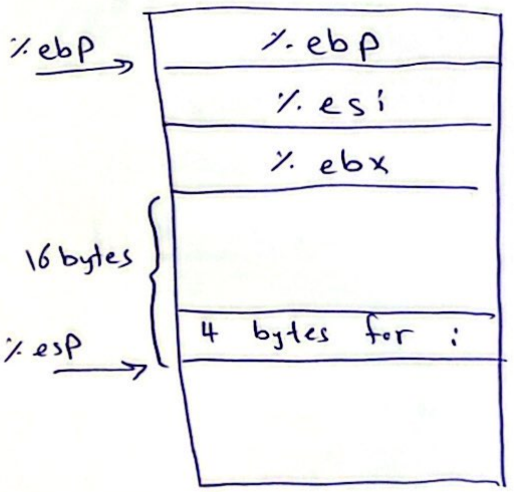
# ~~32~~ % ebx = (32 + %esp)

# value of the memory stored at (%esp + 32) is the value of i

# this line is ~~equivalent to~~ setting the parameter of the function "num" to i (int num = i)

# num in %ebx and it took the value of i which was the argument of the function call

```
testl   %ebx , %ebx

# testl ands the two arguments and sets the condition codes
  without storing the result

# anding the same argument with itself to see if it equals zero
# result of anding ~~with~~ will only equal zero if %ebx value is zero

 jne  L5
# L5 explained in page 9
# jne → jnz   ~ZF   jumps if ZF not equal zero
   So  if  it equals zero :

 movl    $1 , %eax

# move immediate value 1 to %eax to be returned
# then we dig into L1

L1 :
    addl  $24 , %esp
  # increments the stack pointer by 24 which means freeing the
    24 allocated bytes

    popl  %ebx
 # increment %esp by 4
```



```
     ret
   # returns the value stored in %eax which is 1

   # ~~return~~ equivalent to return 1;
```

L5:

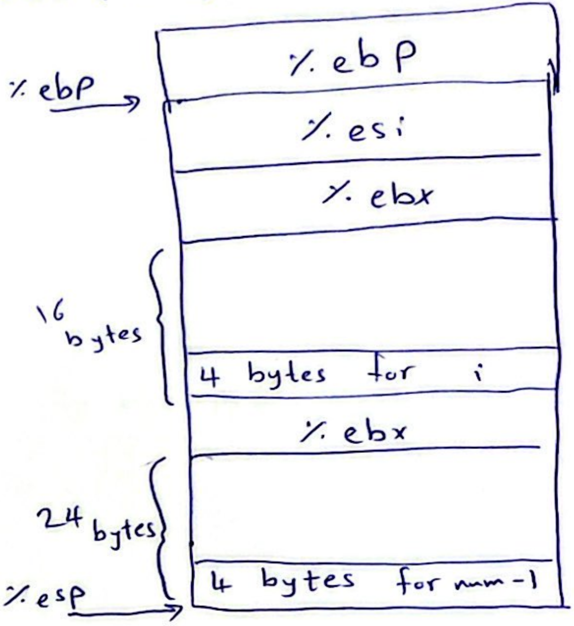
```
leal -1(%ebx), %eax
```

# %eax = %ebx - 1

# %ebx represents num so now %eax holds (num-1)

```
movl %eax, (%esp)
```

# move the value of %eax which is num-1 to the memory location of %esp to be used by the factorial function again



```
call _factorial
```

# now factorial will use the 4 bytes for num-1 as its new "num"

```
imull %ebx, %eax
```

$$\%eax = \underbrace{\%eax}_{\text{result of the Factorial (num-1)}} * \underbrace{\%ebx}_{\text{num}}$$

# equivalent to num * factorial (num-1) and putting the value in %eax to be returned

```
jmp L1
```

# goto L1 to free the stack and return %eax value as explained in page 8