Consider a following C function

1) 2 formal Parameters.
2) 3 local variables.

```
void my_function (int n1, int n2)
{
    int i, j, k
    ___________
    | Body    |
    
}
```

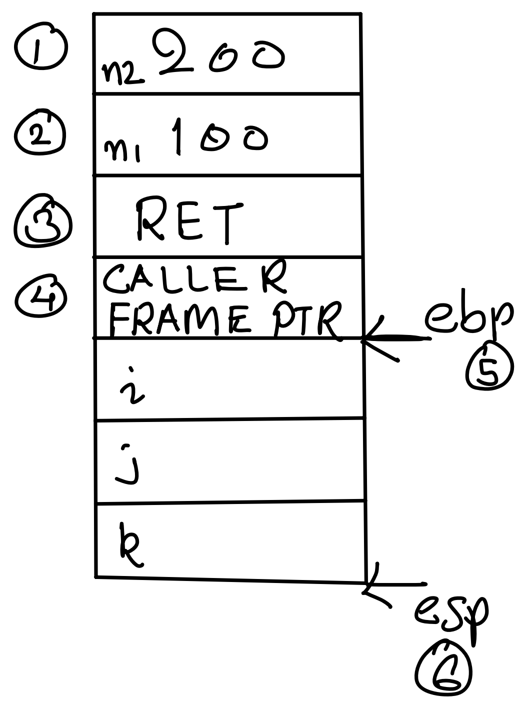
Call to my_function.

my_function (100, 200);

Assembly translation

1. pushl $200
2. pushl $100
3. call my_function.

   addl $8, %esp



```
my_function:
④ pushl %ebp
⑤ movl  %esp, %ebp
⑥ subl  $12, %esp
```

```
    .globl my_function
    .type my_function, @function
my_function:
        pushl %ebp
        movl  %esp, %ebp
        ----------------
        subl  $4n, %esp
        # Body
        movl  %ebp, %esp
        popl  %ebp
        ret
```

n

Function with 1 local variable:

```
# BODY OF SUCH FUNCTION
.globl fun_name
.type fun_name, @function
fun_name:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $4, %esp      # 4 * nr_local_vars == 4 * 1 == 4

    # BODY

    movl   %ebp, %esp
    popl   %ebp
    ret
```

Function with 2 local variables

```
.globl fun_name
.type  fun_name, @function
fun_name:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $8, %esp   # 4 * nr_local_vars == 4 * 2 == 8

    # BODY

    movl   %ebp, %esp
    popl   %ebp
    ret
```

Functions with 3 local variables

```
.globl fun_name
.type  fun_name, @function
fun_name:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $12, %esp     # 4 * nr_local_vars == 4 * 3 == 12

    # BODY

    movl   %ebp, %esp
    popl   %ebp
    ret
```

```
# Generalization:
# Function with n local variables

.globl fun_name
.type  fun_name, @function
fun_name:
   pushl  %ebp
   movl   %esp, %ebp
   subl   $4*n, %esp

   # BODY

   movl   %ebp, %esp
   popl   %ebp
   ret
#------------------------------------------------------------
```

Combo of parameters and local variables:

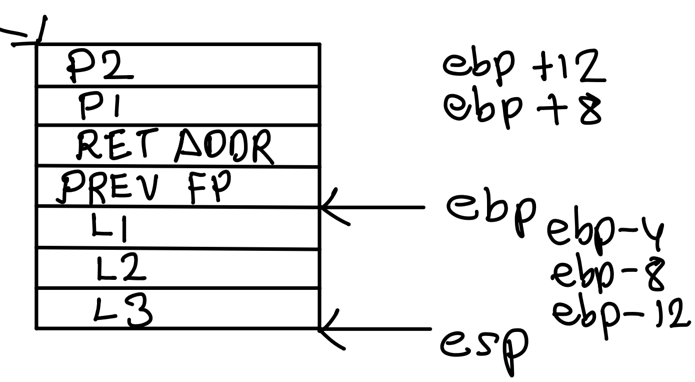
2 parameters, 3 local variables

```
CALL:
   pushl  P2
   pushl  P1
   call   fun_name
   addl   $8, %esp

DEF:
.globl    fun_name
.type     fun_name, @function
fun_name:
   pushl  %ebp
   movl   %esp, %ebp
   subl   $12, %esp

   # BODY

   movl   %ebp, %esp,
   popl   %ebp
   ret
```



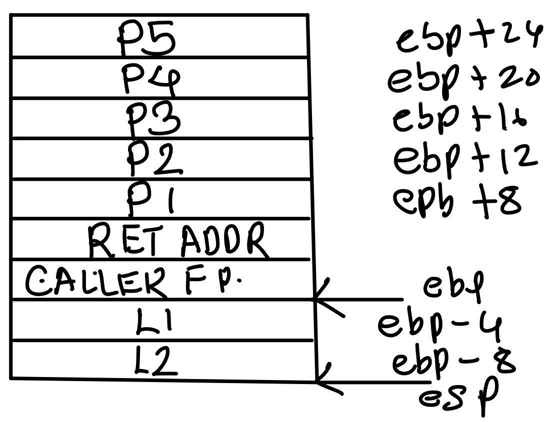
```
# Combination: 5 parameter, 2 local variables:

CALL:
    pushl  P5
    pushl  P4
    pushl  P3
    pushl  P2
    pushl  P1
    call   fun_name
    addl   $20, %esp

DEF:
.globl fun_name
.type  fun_name, @function
fun_name:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $8, %esp

    # BODY

    movl   %ebp, %esp
    popl   %ebp
    ret
```




```
Offset of kth parameter = 8 + (k-1) * 4

Offset of kth local variable = -4*k
```

Generalized version:

A function having 'M' parameters and, 'N' local variables

CALL:

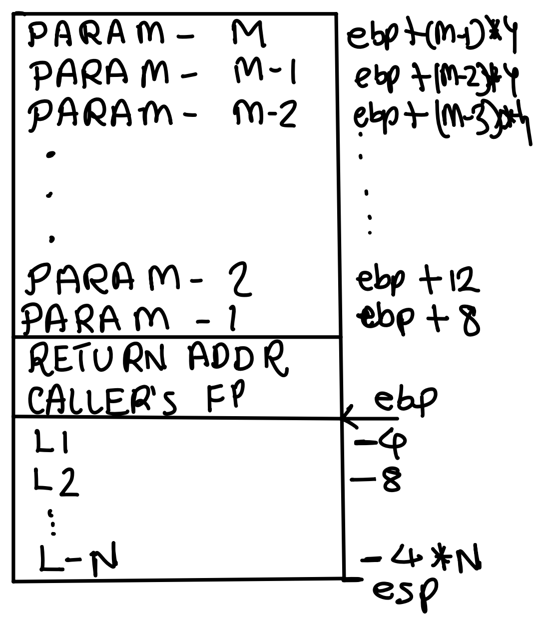
```
pushl  Param-M
pushl  Param-M-1
pushl  Param-M-2
.
.
.
pushl  Param-2
pushl  Param-1
call   function_name
addl   $4*M, %esp
```

DEF:

```
.globl function_name
.type  function_name, @function
function_name:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $4*N, %esp

    # BODY

    movl   %ebp, %esp
    popl   %ebp
    ret
```



Offset of kth Param = 8 + (k-1) * 4 where 1 <= k <= M

Offset of kth Local Variable = -4*k where 1 <= k <= N

```
int my_add(int num1, int num2)
{
   int sum;

   sum = num1 + num2;
   return (sum);
}

# Above function has 2 parameters: num1 and num2, both are ints
# In assembly, we wont be able to refer to parameter memory locations by
# name, instead we must use offsets with respect to ebp.
# assuming that the calling convention of the caller / callee is C calling
# convention, we can determin the offsets of num1 and num2 to be 8 and 12 resp
# Same remarks can be extended towards local variables. The local variable sum
# will not be accessible in assembly by variable name 'sum'. Instead offset
# with respect to ebp must be used. Again, assuming the C calling convention
# between the caller and the callee, we can determine offset of the location
# to be -4
# We can generate assembly code without knowing caller
# Don't forget to keep return value (i.e. summation of params in eax )

.globl my_add
.type  my_add, @function
my_add:
   pushl  %ebp
   movl   %esp, %ebp
   subl   $4, %esp

   movl   8(%ebp), %eax    # eax <- num1 (in C code)
   movl   12(%ebp), %edx   # edx <- num2 (in C code)
   addl   %edx, %eax       # eax == num1 + num2
   movl   %eax, -4(%ebp)   # sum(in C code) <- num1 + num2

   movl   %ebp, %esp
   popl   %ebp
   ret

int my_add(int, int);

int my_add(int x, int y);

int my_add(int a, int b){
}
```

1) Indexed addressing mode
2) How negative numbers are represented inside memory
3) How signed and unsigned data is compared using cmp instruction?
4) How result of the comparison is stored in four bits of eflags register.
   Zero(Z), Carry(C), Sign(S), Overflow(O)
5) How the result of comparison is accessed by jump instructions.
6) Total jump instructions.
7) Branching looping statements (5 branching/3 looping) conversion

```
#-------------------------------------------------------------------------

Calling Convention Internals
   pushl instruction
   popl instruction
   call instruction
   ret instruction
   machine stack manipulation
   with debugger

   Calling convention proof.

#-----------------------------

All C statements -> assembly

#---------------
insertion sort , linked list
#---------------

bit of floating point
#--------------------
Assembly in C code (inline assembly)

char* p = "Hello"; # p[0]

p:
.string   "Hello"

L:
.string   "Hello"

p:
.int   L
```

```
movl   p, %eax
movb   (%eax), %dl
```