

# Arquitecturas de las Computadoras

## TP3 - Assembler y C

### Introducción

Todo programa que corra en una computadora, tiene que estar en un lenguaje que ésta entienda; en código de máquina. El código de máquina hace uso de los registros del procesador, del stack, de las interrupciones, etc. Así que un programa escrito en C, en Assembler, COBOL, Pascal, Ada, etc; a fin de cuentas termina en código de máquina, lo que implica es que siempre se puede hacer una interface para conectar los lenguajes entre sí. En particular, cómo entre código de máquina y Assembler hay una relación de biyectiva, es fácil conectar este lenguaje con cualquier otro, ya que con Assembler hay que respetar el funcionamiento interno del otro lenguaje para poder interactuar.

Es interesante poder hacer una interface con Assembler ya que éste tiene control directo sobre las operaciones del procesador. No como los lenguajes de propósito general que se abstraen de la arquitectura. Con este lenguaje se pueden deshabilitar las interrupciones, habilitar modo protegido, cambiar la dirección del directorio de páginas, modificar el stack pointer, etc.

El lenguaje C es por excelencia el lenguaje de elección para el desarrollo básico de un sistema operativo. Por su bajo nivel, versatilidad, manejo de memoria y su independencia de servicios que necesitan otros lenguajes. Otros lenguajes de programación de más alto nivel, por ejemplo C++, necesitan configuraciones especiales para funcionar.

La convención de cómo realizar las llamadas entre distintos lenguajes, cómo se reserva espacio para las variables, cómo se alinean los datos, etc; es parte de la ***Application Binary Interface (ABI)***

### Llamadas de Assembler a C

Este camino es el más fácil de entender. Si recordamos que al final de cuentas, una función en C es una dirección de memoria, no debería ser difícil hacer una llamada desde ASM a C. Vamos a utilizar el programa ***nasm*** para compilar los archivos en asm, ***gcc*** para compilar los archivos en C y también para linkeditarlos entre sí y contra la biblioteca estándar de C.

### Ejemplo 1

Escribir una función en C que imprima por pantalla "Hola Mundo". Luego, escribir una función en ASM que llame a dicha función

```
; main.asm
GLOBAL main
EXTERN hello_world

section .text

main:
    call hello_world

    mov eax, 1      ; syscall exit()
    mov ebx, 0      ; parametro para exit()
    int 80h
```

```
//hello.c
#include <stdio.h>

void hello_world();

void hello_world() {
    printf ("Hello World!\n");
}
```

Para compilar y correr los archivos anteriores correr las siguientes líneas:

```
$> nasm -f elf32 main.asm
$> gcc -c -m32 hello.c
$> gcc -m32 main.o hello.o -o hello
$> ./hello
```

La directiva **-f elf32** le indica a *nasm* que genere un archivo objeto con formato *elf* (*executable linux format*) para una arquitectura de 32 bits. Algo equivalente hace la directiva **-m32**. Si no se especifica esto, el formato default es el de la máquina actual.

Note que ahora, en lugar de utilizar la etiqueta *\_start* ahora se utiliza la etiqueta *main*. Esto es porque la biblioteca de C ya tiene una etiqueta de entrada, que configura el sistema para luego llamar a nuestro *main*.

**Pregunta:** Ya que ahora estamos en un entorno de C, ¿De qué otra forma se podría haber terminado el programa?

➡ con `return 0`

```
mov eax, 0
ret
```

## Ejemplo 2

En este ejemplo, vamos a pasar un valor a su representación en ASCII numérica.

Recordar cómo es el pasaje de parámetros en C, por Stack en 32 bits: Si una función tiene la siguiente forma:

```
int fnc(arg1, arg2, arg3);
```

Entonces, los argumentos se pasan de derecha a izquierda, de tal forma, que el primer argumento, sea el primero en el stack. Es decir, primero **arg3**, luego **arg2** y por último **arg1**. Luego, el resultado de dicha función se devuelve por el registro `eax`.

La función de C *sprintf* escribe un string con formato en un stream, tiene el siguiente prototipo:

```
int sprintf(char *str, const char *fmt, ...);
```

Esto quiere decir que para llamar a esta función, primero será necesario pushear al stack los argumentos variables, luego la cadena de formato (que le indicará a la función cómo levantar la información) y por último el destino. Luego, se llama a la función *puts* que imprime por pantalla una cadena.

```
int puts(const char *str);
```

### Código:

```
;main.asm
GLOBAL main
EXTERN puts, sprintf

section .rodata
fmt db "%d", 0
number dd 123456790

section .text
main:
    push dword [number]
    push fmt
    push buffer

    call sprintf

    add esp, 3*4
```

```
    push buffer
    call puts

    add esp, 4
    ret

section .bss
buffer resb 40
```

Para compilar este código, simplemente hay que correr las siguientes líneas:

```
$> nasm -f elf32 main.asm
$> gcc -m32 main.o -o main
$> ./main
```

### Preguntas:

para "borrar" del stack los parámetros que se pasan a las funciones {

- ¿Porqué se le suma al stack, luego de llamar a la función `sprintf`, 12 bytes? ¿Porque se hace lo mismo luego de llamar a `puts`?
- ¿Cual es el valor de retorno de la función *main*? ➡ el cero

### Registros a preservar entre llamadas

Entre llamadas de C, hay ciertos registros que las funciones deben preservar para no afectar el funcionamiento de otras funciones. Las funciones que llaman deben tener en cuenta que registros pueden cambiar, de tal forma de hacer un backup antes de llamarlas.

Los registros a preservar entre las llamadas de las funciones son:

- ebx
- esi
- edi
- ebp
- esp

Es decir, cuando una función termina, debe dejar exactamente los registros anteriores como los recibió.

### Stack Frame

Es parte de la ABI de C, cuando se entra a una función, armar esta estructura. Su función es preservar los registros de stack y proveer un sistema estándar de acceso a los parámetros de dicha función.

Con este sistema, la el stack queda de la siguiente forma:

EBP	EBP del contexto anterior
EBP + 4	Dirección de Retorno
EBP + 8	Primer argumento
EBP + 12	Segundo argumento
EBP + (n+1)*4	n-esimo argumento

En C *main* es una función que tiene el siguiente prototipo

```
int main(int argc, char *argv[]);
```

El *program loader*, además de configurar algunas cosas, pasa como parámetro de dicha función los argumentos del programa. Entonces se pueden acceder de la siguiente forma:

```
;main.asm
GLOBAL main
EXTERN printf

section .rodata
fmt db "Cantidad de argumentos: %d\n", 0

section .text
main:
    push ebp        ; Armado de stack frame
    mov ebp, esp    ;

    push dword [ebp+8]
    push fmt
    call printf
    add esp, 2*4

    mov eax, 0
```

```

mov esp, ebp ; Desarmado de stack frame
pop ebp
ret

```

EBP
dir de ret
#args
argv[]

### Ejercicio 1

Modificar el ejemplo anterior, para que además de imprimir la cantidad de argumentos, imprima cada uno de los argumentos.

**⚠ Los argumentos que paso a una función en C los puseo de dar a req**  
**Alineamiento a palabra** Ej. printf ("num%d", 10) ⇒ puse 10 y dsp string

El procesador cada vez que accede a memoria, lo hace leyendo y escribiendo siempre 4 bytes (el tamaño de la palabra en una arquitectura x86).

ESP: 0xDD12

0xDD18  
0xDD14  
0xDD10  
0xDD0C  
0xDD08  
0xDD04  
0xDD00


ESP: 0xDD0C


Cuando el procesador accede a un dato des-alineado, debe hacer dos lecturas, una para pedir la primer porción de los datos, y otra para la segunda porción; es decir, dos accesos. Para la escritura requiere más operaciones, ya que debe leer de memoria para salvar el dato que no va a sobrescribir, luego combinarlo con el dato que si va a escribir para la primera porción, y luego recuperar el dato de la segunda porción para escribir la segunda porción.

Una forma muy fácil de arreglar este problema, es negar los últimos 4 bits. Cómo el stack crece hacia menores valores de esp, seguro esa posición va a

estar por detrás del puntero actual. De esta forma se puede hacer un acceso a memoria más eficiente.

```
GLOBAL main

ALIGN 4

main:
    push ebp
    mov ebp, esp

    ; declaración de variables

    and esp, -16

    ; ... programa

    mov esp, ebp
    pop ebp
    ret
```

**Responder:** ¿Cual es la representación de -16 en hexadecimal? ➡ **FFFF FFF0h**

## Ejercicio 2

Escriba un programa en C, que realice lo mismo que el ejercicio anterior (imprimir por salida estándar la cantidad de argumentos del programa),

```
// arguments.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Cantidad de argumentos %d\n", argc);
    return 0;
}
```

Compilarlo con el siguiente comando (que elimina un montón de elementos de control de gcc):

```
$> gcc -c arguments.c -m32 -fno-dwarf2-cfi-asm -fno-exceptions -S -fno-asynchronous-unwind-tables -masm=intel
```

Compare la salida del archivo *arguments.s* con el ejemplo provisto en ASM. Más allá de los elementos de control, ¿Que diferencia encuentra? ¿Qué similitudes?

Para pensar: ¿Se imagina porque no se deben comparar las estructuras por valor, sinó por miembro por miembro? ¿Porqué habría *basura* entre sus componentes?

## Llamadas de C a Assembler

Este es el camino que tiene una aplicación más práctica. ¿Qué punto tiene programar en Assembler cuando se tiene un lenguaje eficiente, práctico y liviano? Salvo que exista una razón muy puntual; cómo que no exista un compilador; evitaremos programar en Assembler.

Nuevamente, las funciones a fin de cuentas son direcciones, así que es posible llamar funciones desde C hacia Assembler.

### Ejemplo

Si no contamos con una biblioteca estándar que acceda a la API del SO para enviar o recibir información, debería ser implementado por nosotros. Ya vimos que hacer esto completamente en Assembler, ahora lo tendremos que hacer nosotros.

El PID (Process ID) es un número que utiliza Linux para identificar los procesos que están corriendo. Si se corre el comando **ps x**, se puede ver una lista de los procesos que están corriendo actualmente que son del usuario. Obviamente, como **ps** es un proceso, también aparece en la lista.

No hay ninguna función estándar que informe cual es el **pid** de un programa. Cómo es información que maneja el OS, habrá que pedírsela a él.

```
//main.c
#include<stdio.h>

unsigned int pid();
```



```
int main(int argc, char *argv[]) {
    int mpid = pid();
    printf("Process Id: %d\n", mpid);
    return 0;
}
```

```
; libasm.asm
GLOBAL pid

pid:
    push ebp
    mov ebp, esp

    mov eax, 0x14 ; syscall getpid
    int 0x80
    ; el resultado ya está en eax

    mov esp, ebp
    pop ebp
    ret
```

Compilar y correr el código de ejemplo con las siguientes líneas:

```
$> nasm -f elf32 libasm.asm
$> gcc -c -m32 main.c
$> gcc -m32 main.o libasm.o -o pid
$> ./pid
```

Ejecútelo varias veces para comprobar que el **pid** va en aumento.

Compile el archivo *main.c* de tal forma de obtener su salida en Assembler, compruebe que efectivamente se hace una llamada a la función **pid()** mediante la línea.

```
call pid
```

Para que C sepa de qué forma, con que argumentos y con qué tipo de retorno se va a llamar a una función, es necesario especificar su prototipo, por eso se escribió antes de main la línea:

```
unsigned int pid();
```

Esto le indica al compilador que no tiene que chequear, ni mandar argumentos a pid para invocarla, y que luego, el tipo de datos que tendrá en *eax* será de tipo *int* sin signo. Recordar que en ASM no hay tipo de datos!

## Recepción de Parámetros

También es necesario poder acceder a los argumentos de las funciones, por ejemplo si se implementara la función puts en una biblioteca de funciones, sería algo equivalente a:

```
//libc.c
#define STDOUT 1

int sys_write(int fd, void *buffer, int size);

int puts(const char* str) {
    int len = strlen(str);
    return sys_write(STDOUT, (void *) str, len);
}
```

Y la implementación de la función sys\_write podría ser algo cómo:

```
; libasm.asm
GLOBAL sys_write

ALIGN 4
sys_write:
    push ebp
    mov ebp, esp

    push ebx ;preservar ebx

    mov eax, 0x4
    mov ebx, [ebp+8] ; fd
    mov ecx, [ebp+12] ; buffer
    mov edx, [ebp+16] ; length
    int 0x80
```

```
pop ebx

mov esp, ebp
pop ebp
ret
```

### Ejercicio 3

Realice una implementación para cumplir con el ejemplo. Utilice la directiva **-fno-builtin** para indicarle a C que no debe linkeditar contra la biblioteca estándar.

### Variables

Hasta ahora, siempre que hizo falta reservamos espacio en la zona de datos de un programa. Por ejemplo, para guardar una variable. Pero, ¿Qué ocurriría en el caso de que una función sea recursiva?

¿Cómo hace C a que las variables automáticas sólo vivan en el scope de la función? La respuesta es guardar esas variables en el stack, así, cuando un stack frame se destruye, al mismo tiempo se libera la memoria utilizada para sus variables.

### Ejemplo 3

```
//suma.c

int suma(int a, int b) {
    int resultado;
    resultado = a + b;
    return resultado;
}
```

Una posible salida en Assembler podría ser:

```
suma:
    push ebp
    mov  ebp, esp
    sub  esp, 16 ; reservar espacio para las variables
    mov  eax, DWORD PTR [ebp+12] ➡ paso la variable a
    mov  edx, DWORD PTR [ebp+8] ➡ paso la variable b
    add  eax, edx
    mov  [ebp-4], eax ; resultado ➡ asigno a resultado el valor a + b
    mov  eax, [ebp-4] ➡ xq retorno eax

    leave ; otra forma de desarmar el stack frame
    ret
```

De esta forma, la variable resultado, existe únicamente para la función, no tiene visibilidad para una función externa.

#### Reponder:

- ¿Porqué no se debe devolver un arreglo local a una función cómo valor de retorno?
- ¿Porque restan 16 bytes de ESP y no simplemente 4?
- ¿Porqué luego, de haber obtenido el resultado de la suma, que queda en el registro EAX, se lo guarda en la zona de memoria asignada a la variable resultado y luego se la vuelve a leer de ese lugar?

#### Ejemplo 4

Genere la salida en Assembler con GCC de la siguiente función:

```
//factorial.c
int factorial(int n) {
    if (n == 0)
        return 1;
    int factorial_n_1 = factorial(n-1);
    return n*factorial_n_1;
}
```

Identifique a que dirección de memoria relativa corresponde la variable *factorial\_n\_1*.

⇒ **Factorial\_n\_1 esta en EBP -40 ⇒ esta en el stack**

Ahora es obvio que se puede llamar de manera recursiva, ya que cada invocación de la función tendrá su propia versión de dicha variable.

## Ejercicios

#### Ejercicio 5

Escriba una función en Assembler que retorne siempre el valor 7. Llámela con C y muestre por salida estándar dicho valor.

#### Ejercicio 6

Enumere las diferencias de declarar las variables y utilizarlas de las siguientes formas.

a.

```
//variables1.c
int foo() {
    int numero; ➡ Sin inicializar (local)
}
```

➡ Variables locales se guardan en el stack ➡ cuando salgo de la función, "deja de existir"

b.

```
//variables2.c
int foo() {
    int numero = 21; ➡ con inicialización (local)
}
```

⚠ Todas las variables que se reinician con cada llamada a la función se guardan en el Stack sino se guardan en la tabla.

c.

```
//variables3.c
int numero;
int foo() {
    numero = 21; ➡ sin inicializar y luego realizando una escritura (global)
}
```

d.

```
int foo() {
    static int numero = 21; ➡ con inicialización y static i.e. conserva su valor a lo largo de toda la ejecución del programa
}
```

➡ las static están definidas como las globales

e.

```
extern int numero; ➡ variable externa definida en otro archivo
int foo() {
    numero = 10; ⚠ No reserva espacio. El compilador busca la variable en otro archivo (en bar()).
}
```

```
int numero = 21;
int bar() {
    numero = 30; ➡ con inicialización y luego realizando una escritura (global)
}
```

f.

```
static1 int numero = 10;
int foo() {
    ➡ variable global static inicializado y luego realizando una escritura
```

<sup>1</sup> reserva una variable global PERO solo está disponible en dicho archivo ➡ NO puedo usar el EXTERN ⚠

```
numero = 20;  
}
```

### Ejercicio 7

Declare en una función de C, un arreglo de las siguientes formas:

- Sin inicializar
- Con inicialización (`int numeros[20] = {0}`, ó `char msg[] = "mensaje"`)
- Sin inicializar y luego realizando una escritura en el índice 10
- Con inicialización y luego realizando una escritura en el índice 10
- De manera global, sin inicializar
- De manera global inicializando.

Primero suponga cómo debería hacer C para su declaración, y luego compare sus suposiciones con la salida en assembler.

### Ejercicio 8

Suponga que ud. está implementando la biblioteca estándar de C. Basándose en la guía anterior, prepare el contexto para llamar a la función inicial (`main`), y luego termine el programa con el valor de retorno que haya indicado la función `main`. Su *loader* debe utilizar la etiqueta `_start`

### Ejercicio 9

Cómo parte de la implementación de su biblioteca estándar de C, implemente las funciones que le permitan acceder al sistema operativo para:

- Cerrar un programa
- Leer de un file descriptor
- Escribir a un file descriptor (ya implementado)
- Abrir un archivo y cerrarlo

Lista de system calls:

[http://docs.cs.up.ac.za/programming/asm/derick\\_tut/syscalls.html](http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html),  
<http://syscalls.kernelgrok.com/>

Escriba un programa que haga uso de su nueva biblioteca para abra un archivo, imprima línea por línea con el número de línea a la derecha y luego lo cierre. Recuerde compilar con la directiva **-fno-builtin**

## Ejercicio 10

Escriba un programa que imprima por salida estándar el fabricante del procesador. Investigue la función *cpuid* de assembler.

## Ejercicio 11

- Implemente la función en C `fibonacci(n)`, que reciba un número, y devuelva el número de fibonacci que corresponda.
- Compile a mano, una posible salida en assembler de dicho programa.
- Compruebe su resultado con el de haber generado el código con **gcc**
- Haga un seguimiento, paso a paso, de la pila por cada instrucción de assembler ejecutada para un fibonacci de 3.

## Ejercicio 12

Dado el siguiente programa:

```
;ejemplo.s
main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 32
    mov     DWORD PTR [esp+19], 1819043176
    mov     DWORD PTR [esp+23], 1870078063
    mov     DWORD PTR [esp+27], 174353522
    mov     BYTE PTR [esp+31], 0
    lea     eax, [esp+19]
    mov     DWORD PTR [esp], eax
    call    magia
    lea     eax, [esp+19]
    mov     DWORD PTR [esp], eax
    call    printf
    mov     eax, 0
    leave

magia:
    push    ebp
    mov     ebp, esp

    sub     esp, 16
    jmp     .L4
.L6:
    mov     eax, DWORD PTR [ebp+8]
    movzx   eax, BYTE PTR [eax]
    cmp     al, 96
```

```

jle .L5
mov eax, DWORD PTR [ebp+8]
movzx eax, BYTE PTR [eax]
cmp al, 122
jg .L5
mov eax, DWORD PTR [ebp+8]
movzx eax, BYTE PTR [eax]
mov BYTE PTR [ebp-1], al
movzx eax, BYTE PTR [ebp-1]
sub eax, 32
mov BYTE PTR [ebp-1], al
3mov eax, DWORD PTR [ebp+8]
movzx edx, BYTE PTR [ebp-1]
mov BYTE PTR [eax], dl
.L5:
add DWORD PTR [ebp+8], 1
.L4:
mov eax, DWORD PTR [ebp+8]
movzx eax, BYTE PTR [eax]
test al, al ➡ da cero SOLO cuando al vale cero ➡ cuando finaliza el string
jne .L6
leave
ret

```

Indique lo que está haciendo, haga un seguimiento de la pila.

➡ pasa "hello world" a mayúscula

### Ejercicio 13

Deduzca cómo se pasan las estructuras entre funciones, ¿Que diferencia hay entre pasar una estructura por referencia o por copia? ¿Cómo es el retorno?

## x86\_64 ABI

Hasta ahora trabajamos con la ABI de 32 bits de C y Assembler. Básicamente los conceptos son los mismos, lo único que cambia es el pasaje de parámetros.

El pasaje de parámetros se realiza de la siguiente forma:

- Se cargan los argumentos en los registros
- Se llama a la función
- Los argumentos se copian al stack y se referencian desde ahí.

Clasificación:

Los argumentos se clasifican de la siguiente forma:

- INTEGER: char, short, int, long, long long y punteros
- SSE: floats y doubles



- MEMORY: datos mayores a un quadword (8 bytes) y datos desalineados

Y se pasan de la siguiente forma:

Si el dato es INTEGER,  
se van ocupando los registros rdi, rsi, rdx, rcx, r8 y r9 en orden  
Si el dato es SSE,  
se van ocupando los registros xmm0 a xmm7 en orden  
Si el dato es MEMORY, se pasan por stack y devuelven por stack, de izquierda a derecha (igual que en 32 bits)

Para devolver los valores,

Si el dato es INTEGER,  
se utiliza rax y rdx  
Si el dato es SSE,  
se retorna por xmm0 y xmm1

Los registros a preservar son:

- rbp
- rsp
- rbx
- r12
- r13
- r15

Esto implica que hay funciones que aunque tengan distinto prototipo, tienen la misma forma de pasar parámetros, por ejemplo las funciones

```
int foo(int arg1, double arg2);
```

tiene el mismo prototipo que la función:

```
int bar(double arg2, int arg1);
```

En síntesis:

```

typedef struct {
    int a, b;
    double d;
} structparm;
structparm s;
int e, f, g, h, i, j, k;
long double ld;
double m, n;
__m256 y;

extern void func (int e, int f,
                  structparm s, int g, int h,
                  long double ld, double m,
                  __m256 y,
                  double n, int i, int j, int k);

func (e, f, s, g, h, ld, m, y, n, i, j, k);

```

Los registros van a quedar de la siguiente forma:

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rdi: e	%xmm0: s.d	0: ld
%rsi: f	%xmm1: m	16: j
%rdx: s.a, s.b	%ymm2: y	24: k
%rcx: g	%xmm3: n	
%r8: h		
%r9: i		

## Ejercicios

Realice los ejercicios de pasaje de parámetros, pero esta vez en 64 bits.

Para compilar, reemplazar los argumentos **elf32** por **elf64** de **nasm** y **-m32** por **-m64** en gcc.

⇒ **gcc -m64 -no-pie**

## Ejercicios de debugging

Tome los archivos Ejxx y Ejyy disponibles en el campus, estos son programas assembler que usan funciones de libreria estandard de C. Compile estos programas en 32 bits.

El programa Ejxx debiera imprimir todos los argumentos que recibe. Ejecute el programa Ejxx con y sin argumentos. Vera que cuando recibir argumentos el programa cancela. Diagnostique el problema y solucione el código.

EL Ejyy debiera imprimir el número 1234567890 pero tambien falla. Diagnostique el problema y solucione el codigo.

## Ejercicio 2 - Llamada de ASM a C

```
GLOBAL main
EXTERN printf
```

```
section .rodata
fmt db "Cantidad de argumentos: %d", 10, 0
```

```
section .text
```

```
main:
```

```
1 push ebp
```

```
2 mov ebp, esp
```

```
3 push dword [ebp+8]
```

```
4 push fmt
```

```
5 call printf
```

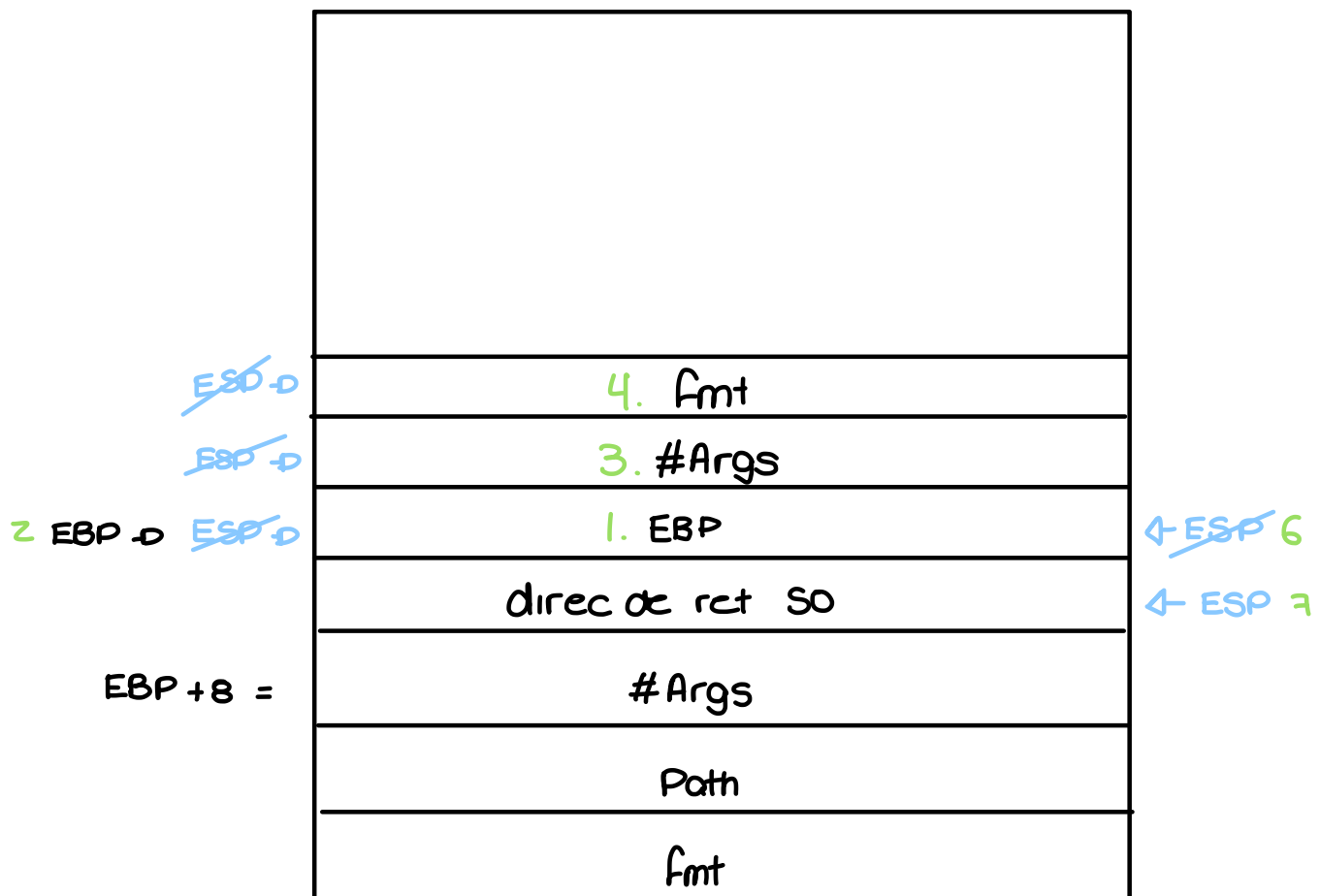
```
6 add esp, 2*4 → "borra" los parametros de la pila
```

```
7 mov eax, 0
```

```
8 mov esp, ebp
```

```
9 pop ebp
```

```
ret
```



## Ejercicio 2 - Salida en ASM

```
.file "arguments.c"
.intel_syntax noprefix
.text
.section .rodata
.LC0:
.string "Cantidad de argumentos %d\n"
.text
.globl main
.type main, @function
main:
endbr32
    lea ecx, 4[esp] ➡ [esp + 4]
    and esp, -16 ➡ load effective address y cargo en ecx la dir a la #Args
    push DWORD PTR -4[ecx] ➡ [ecx - 4] = dir de ret al SO
    push ebp
    mov ebp, esp
    push ebx
    push ecx
    call __x86.get_pc_thunk.ax
    add eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    mov edx, ecx ➡ en ecx esta la dir a #Args
    sub esp, 8
    push DWORD PTR [edx] ➡ pushea #Args
    lea edx, .LC0@GOTOFF[eax] ➡ carga en eax la dir al string
    push edx ➡ pushea la dir al string
    mov ebx, eax
    call printf@PLT ➡ llamo a printf
    add esp, 16 ➡ "borra" los parametros
    mov eax, 0
    lea esp, -8[ebp]
    pop ecx
    pop ebx
    pop ebp
    lea esp, -4[ecx] ➡ esp vuelve a apuntar a la direccion de retorno al SO
    ret
.size main, .-main
.section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comdat
.globl __x86.get_pc_thunk.ax
.hidden __x86.get_pc_thunk.ax
.type __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
    mov eax, DWORD PTR [esp]
    ret
.ident "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 4
.long 1f - 0f
.long 4f - 1f
.long 5
0:
1:
    .string "GNU"
```

```
        .align 4
        .long    0xc0000002
        .long    3f - 2f
2:      .long    0x3
3:
        .align 4
4:
```

## Ejercicio 4 - Salida en ASM

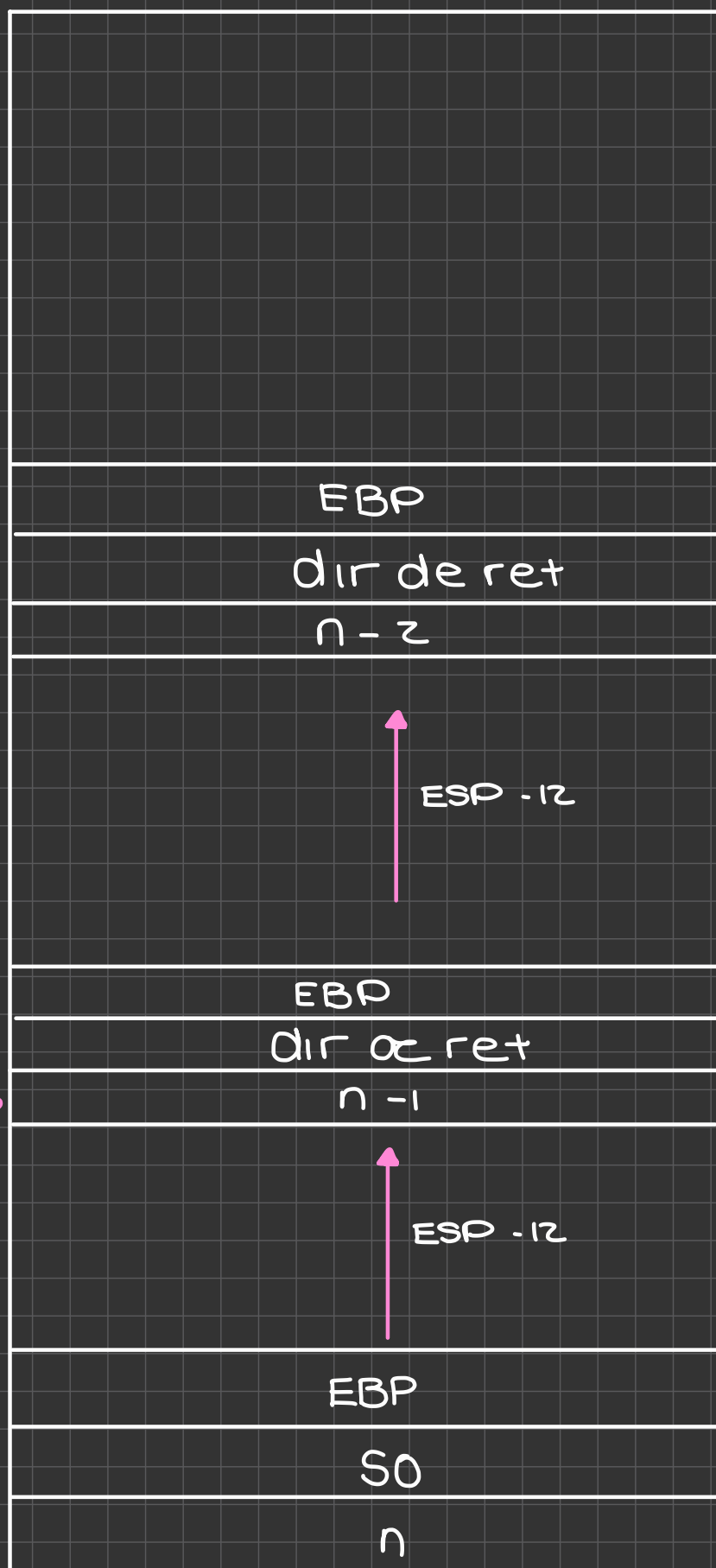
```
.file "factorial.c"
.intel_syntax noprefix
.text
.globl factorial
.type factorial, @function
factorial:
    endbr32
    push    ebp
    mov     ebp, esp
    sub     esp, 24
    call    __x86.get_pc_thunk.ax
    add     eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    cmp     DWORD PTR 8[ebp], 0
    jne     .L2
    mov     eax, 1
    jmp     .L3
.L2:
    mov     eax, DWORD PTR 8[ebp]
    sub     eax, 1
    sub     esp, 12
    push    eax
    call    factorial
    add     esp, 16
    mov     DWORD PTR -12[ebp], eax
    mov     eax, DWORD PTR 8[ebp]
    imul    eax, DWORD PTR -12[ebp]
.L3:
    leave
    ret
.size factorial, .-factorial
.section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comdat
.globl __x86.get_pc_thunk.ax
.hidden __x86.get_pc_thunk.ax
.type __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
    mov     eax, DWORD PTR [esp]
    ret
.ident "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 4
.long 1f - 0f
.long 4f - 1f
.long 5
0:
.string "GNU"
1:
.align 4
.long 0xc0000002
.long 3f - 2f
2:
.long 0x3
3:
.align 4
4:
```

## Ejercicio 4

Cuando  $n - x$  sea cero, guardo un 1 en `eax` y retorno.  
Desarmo el stack y en `[EBP - 12]` guardo `eax` y a `eax` le pongo  $n - x + 1 = [EBP - 8]$   
Multiplico `eax` por `[EBP - 12]` y retorno.

ESP + 16  
⇒ no hace falta  
porque `dsp` se  
desarma el stack

pushes  $n-1$  para llamar ⇒  
a `factorial(n-1)`





## Ejercicio 7 - Sin inicializar

```
.file "foo.c"
.intel_syntax noprefix
.text
.globl foo
.type foo, @function

foo:
    endbr32
    push    ebp
    mov     ebp, esp
    sub     esp, 56
    call    __x86.get_pc_thunk.ax
    add     eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    mov     eax, DWORD PTR gs:20
    mov     DWORD PTR -12[ebp], eax
    xor     eax, eax
    mov     eax, 0
    mov     edx, DWORD PTR -12[ebp]
    xor     edx, DWORD PTR gs:20
    je      .L3
    call    __stack_chk_fail_local

.L3:
    leave
    ret
.size foo, .-foo
.section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comdat
.globl __x86.get_pc_thunk.ax
.hidden __x86.get_pc_thunk.ax
.type __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
    mov     eax, DWORD PTR [esp]
    ret
.hidden __stack_chk_fail_local
.ident "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 4
.long 1f - 0f
.long 4f - 1f
.long 5

0:
.string "GNU"
1:
.align 4
.long 0xc0000002
.long 3f - 2f
2:
.long 0x3
3:
.align 4
4:
```

```
int foo() {
    int numero[10];
    return 0;
}
```

## Ejercicio 7 - Con inicialización

```
.file "foo.c"
.intel_syntax noprefix
.text
.globl foo
.type foo, @function

foo:
    endbr32
    push    ebp
    mov     ebp, esp
    push    edi
    sub     esp, 52
    call    __x86.get_pc_thunk.ax
    add     eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    mov     eax, DWORD PTR gs:20
    mov     DWORD PTR -12[ebp], eax
    xor     eax, eax
    lea     edx, -52[ebp]
    mov     eax, 0
    mov     ecx, 10
    mov     edi, edx
    rep stosd
    mov     eax, 0
    mov     edx, DWORD PTR -12[ebp]
    xor     edx, DWORD PTR gs:20
    je      .L3
    call    __stack_chk_fail_local

.L3:
    add     esp, 52
    pop     edi
    pop     ebp
    ret
    .size   foo, .-foo
    .section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comdat
    .globl  __x86.get_pc_thunk.ax
    .hidden __x86.get_pc_thunk.ax
    .type   __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
    mov     eax, DWORD PTR [esp]
    ret
    .hidden __stack_chk_fail_local
    .ident  "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
    .section .note.GNU-stack,"",@progbits
    .section .note.gnu.property,"a"
    .align 4
    .long   1f - 0f
    .long   4f - 1f
    .long   5

0:
    .string "GNU"

1:
    .align 4
    .long   0xc0000002
    .long   3f - 2f
```

```
2:
    .long    0x3
3:
    .align 4
4:
```

```
int foo(){
    int numero[10] = {0};
    return 0;
}
```

## Ejercicio 7 - Sin inicializar y luego reescritura

```
.file "foo.c"
.intel_syntax noprefix
.text
.globl foo
.type foo, @function
foo:
    endbr32
    push    ebp
    mov     ebp, esp
    sub     esp, 56
    call    __x86.get_pc_thunk.ax
    add     eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    mov     eax, DWORD PTR gs:20
    mov     DWORD PTR -12[ebp], eax
    xor     eax, eax
    mov     DWORD PTR -12[ebp], 1
    mov     eax, 0
    mov     edx, DWORD PTR -12[ebp]
    xor     edx, DWORD PTR gs:20
    je      .L3
    call    __stack_chk_fail_local
.L3:
    leave
    ret
.size foo, .-foo
.section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comdat
.globl __x86.get_pc_thunk.ax
.hidden __x86.get_pc_thunk.ax
.type __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
    mov     eax, DWORD PTR [esp]
    ret
.hidden __stack_chk_fail_local
.ident "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 4
.long 1f - 0f
.long 4f - 1f
.long 5
0:
.string "GNU"
1:
.align 4
.long 0xc0000002
.long 3f - 2f
2:
.long 0x3
3:
.align 4
4:
```

```
int foo() {
    int numero[10];
    numero[10] = 1;
    return 0;
}
```

## Ejercicio 1 - Con inicialización y luego reescritura

```
.file "foo.c"
.intel_syntax noprefix
.text
.globl foo
.type foo, @function

foo:
    endbr32
    push    ebp
    mov     ebp, esp
    push    edi
    sub     esp, 52
    call    __x86.get_pc_thunk.ax
    add     eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    mov     eax, DWORD PTR gs:20
    mov     DWORD PTR -12[ebp], eax
    xor     eax, eax
    lea     edx, -52[ebp]
    mov     eax, 0
    mov     ecx, 10
    mov     edi, edx
    rep stosd
    mov     DWORD PTR -52[ebp], 1
    mov     DWORD PTR -12[ebp], 2
    mov     eax, 0
    mov     edx, DWORD PTR -12[ebp]
    xor     edx, DWORD PTR gs:20
    je      .L3
    call    __stack_chk_fail_local

.L3:
    add     esp, 52
    pop     edi
    pop     ebp
    ret
    .size   foo, .-foo
    .section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comdat
    .globl  __x86.get_pc_thunk.ax
    .hidden __x86.get_pc_thunk.ax
    .type   __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
    mov     eax, DWORD PTR [esp]
    ret
    .hidden __stack_chk_fail_local
    .ident  "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
    .section .note.GNU-stack,"",@progbits
    .section .note.gnu.property,"a"
    .align 4
    .long   1f - 0f
    .long   4f - 1f
    .long   5

0:
    .string "GNU"

1:
    .align 4
```

```
        .long    0xc0000002
        .long    3f - 2f
2:
        .long    0x3
3:
        .align 4
4:
```

```
int foo(){
    int numero[10] = {1};
    numero[10] = 2;
    return 0;
}
```

## Ejercicio 7 - De manera global sin inicializador

```
.file "foo.c"
.intel_syntax noprefix
.text
.comm numero,40,32
.globl foo
.type foo, @function

foo:
    endbr32
    push    ebp
    mov     ebp, esp
    call    __x86.get_pc_thunk.ax
    add     eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    mov     eax, DWORD PTR numero@GOT[eax]
    mov     DWORD PTR 40[eax], 1
    mov     eax, 0
    pop     ebp
    ret
    .size   foo, .-foo
    .section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.g
et_pc_thunk.ax,comdat
    .globl  __x86.get_pc_thunk.ax
    .hidden __x86.get_pc_thunk.ax
    .type   __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
    mov     eax, DWORD PTR [esp]
    ret
    .ident  "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
    .section .note.GNU-stack,"",@progbits
    .section .note.gnu.property,"a"
    .align 4
    .long   1f - 0f
    .long   4f - 1f
    .long   5
0:
    .string "GNU"
1:
    .align 4
    .long   0xc0000002
    .long   3f - 2f
2:
    .long   0x3
3:
    .align 4
4:
```

```
int numero[10];
int foo() {
    numero[10] = 1;
    return 0;
}
```

## Ejercicio 7 - De manera global con inicialización

```
.file "foo.c"
.intel_syntax noprefix
.text
.globl numero
.bss
.align 32
.type numero, @object
.size numero, 40
numero:
.zero 40
.text
.globl foo
.type foo, @function
foo:
endbr32
push ebp
mov ebp, esp
call __x86.get_pc_thunk.ax
add eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
mov DWORD PTR numero@GOTOFF[eax+40], 1
mov eax, 0
pop ebp
ret
.size foo, .-foo
.section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comdat
.globl __x86.get_pc_thunk.ax
.hidden __x86.get_pc_thunk.ax
.type __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
mov eax, DWORD PTR [esp]
ret
.ident "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 4
.long 1f - 0f
.long 4f - 1f
.long 5
0:
.string "GNU"
1:
.align 4
.long 0xc0000002
.long 3f - 2f
2:
.long 0x3
3:
.align 4
4:
```

```
int numero[10] = {0};
int foo() {
    numero[10] = 1;
    return 0;
}
```



## Ejercicio 11 - Salida en asm

```
.file    "fibonacciC.c"
.intel_syntax noprefix
.text
.globl   fibonacci
.type    fibonacci, @function
fibonacci:
    endbr32
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 4
    call    __x86.get_pc_thunk.ax
    add     eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    cmp     DWORD PTR 8[ebp], 0
    jne     .L2
    mov     eax, 0
    jmp     .L3
.L2:
    cmp     DWORD PTR 8[ebp], 1
    jne     .L4
    mov     eax, 1
    jmp     .L3
.L4:
    mov     eax, DWORD PTR 8[ebp]
    sub     eax, 1
    sub     esp, 12
    push    eax
    call    fibonacci
    add     esp, 16
    mov     ebx, eax
    mov     eax, DWORD PTR 8[ebp]
    sub     eax, 2
    sub     esp, 12
    push    eax
    call    fibonacci
    add     esp, 16
    add     eax, ebx
.L3:
    mov     ebx, DWORD PTR -4[ebp]
    leave
    ret
.size     fibonacci, .-fibonacci
.section .rodata
.LC0:
.string   "Fibonacci de: "
.LC1:
.string   "%d"
.LC2:
.string   "%d\n"
.text
.globl   main
.type    main, @function
main:
    endbr32
```

```

    lea    ecx, 4[esp]
    and    esp, -16
    push   DWORD PTR -4[ecx]
    push   ebp
    mov    ebp, esp
    push   ebx
    push   ecx
    sub    esp, 16
    call   __x86.get_pc_thunk.bx
    add    ebx, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    mov    eax, DWORD PTR gs:20
    mov    DWORD PTR -12[ebp], eax
    xor    eax, eax
    sub    esp, 12
    lea    eax, .LC0@GOTOFF[ebx]
    push   eax
    call   printf@PLT
    add    esp, 16
    sub    esp, 8
    lea    eax, -20[ebp]
    push   eax
    lea    eax, .LC1@GOTOFF[ebx]
    push   eax
    call   __isoc99_scanf@PLT
    add    esp, 16
    mov    eax, DWORD PTR -20[ebp]
    sub    esp, 12
    push   eax
    call   fibonacci
    add    esp, 16
    mov    DWORD PTR -16[ebp], eax
    sub    esp, 8
    push   DWORD PTR -16[ebp]
    lea    eax, .LC2@GOTOFF[ebx]
    push   eax
    call   printf@PLT
    add    esp, 16
    mov    eax, 0
    mov    edx, DWORD PTR -12[ebp]
    xor    edx, DWORD PTR gs:20
    je     .L7
    call   __stack_chk_fail_local
.L7:
    lea    esp, -8[ebp]
    pop    ecx
    pop    ebx
    pop    ebp
    lea    esp, -4[ecx]
    ret
.size    main, .-main
.section .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.g
et_pc_thunk.ax,comdat
.globl   __x86.get_pc_thunk.ax
.hidden __x86.get_pc_thunk.ax

```

```

        .type      __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
        mov        eax, DWORD PTR [esp]
        ret
        .section .text.__x86.get_pc_thunk.bx,"axG",@progbits,__x86.g
et_pc_thunk.bx,comdat
        .globl     __x86.get_pc_thunk.bx
        .hidden    __x86.get_pc_thunk.bx
        .type      __x86.get_pc_thunk.bx, @function
__x86.get_pc_thunk.bx:
        mov        ebx, DWORD PTR [esp]
        ret
        .hidden    __stack_chk_fail_local
        .ident      "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
        .section .note.GNU-stack,"",@progbits
        .section .note.gnu.property,"a"
        .align 4
        .long       1f - 0f
        .long       4f - 1f
        .long       5
0:
        .string     "GNU"
1:
        .align 4
        .long       0xc0000002
        .long       3f - 2f
2:
        .long       0x3
3:
        .align 4
4:

```

```

#include <stdio.h>

int fibonacci(int n) {
    if(n == 0) {
        return 0;
    }
    if(n == 1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}

int main(void) {
    int n;
    printf("Fibonacci de: ");
    scanf("%d", &n);
    int res = fibonacci(n);
    printf("%d\n", res);
    return 0;
}

```

```

section .text
GLOBAL main
EXTERN printf

main:
    push ebp
    mov ebp, esp
    push 3
    call fibo
    add esp, 4
    push eax
    push fmt
    call printf
    add esp, 8
    mov eax, 0
    mov esp, ebp
    pop ebp
    ret

fibo:
    push ebp
    mov ebp, esp
    mov ebx, [ebp + 8]
    mov eax, 0
    mov ecx, 0
    mov edx, 1
    .while:
        cmp ebx, 0
        jz .isZero
        cmp ebx, 1
        jz .isOne
        dec ebx
        push ebx
        call fibo
        add esp, 4
        mov eax, ecx
        add eax, edx
        mov ecx, edx
        mov edx, eax
        leave
        ret
    .isOne:
        add eax, 1
        leave
        ret
    .isZero:
        add eax, 0
        leave
        ret

```

```

section .rodata
fmt db "%d", 10, 0

```

- $ebx = 3 = z = 1$
- $eax = 0 = 1 = 1 = z$
- $ecx = 0 = 1 = 1$
- $edx = 1 = 1 = z$

EBP
ret
ebx = 1
EBP
ret
ebx = z
EBP
ret ; fmt
3 ; eax = z
EBP

# Ejercicio 12

SUB ESP, 16 →

- <sup>1</sup> ESP + 19
- <sup>2</sup> ESP + 23
- <sup>3</sup> ESP + 27

## Observaciones:

- 1819043176 = 6C6C6568h
- 1870078063 = 6F77206Fh
- 174353522 = 0A646C72h

ebp			
ret			
68	65	6C	6C
			<sup>1</sup> 68
65	6C	6C	<sup>2</sup> 6F
20	77	6F	<sup>3</sup> 72
6C	64	0A	∅
ebp			
ret			