# Pila
# Convención C
# Interacción C-ASM

Organización del Computador II

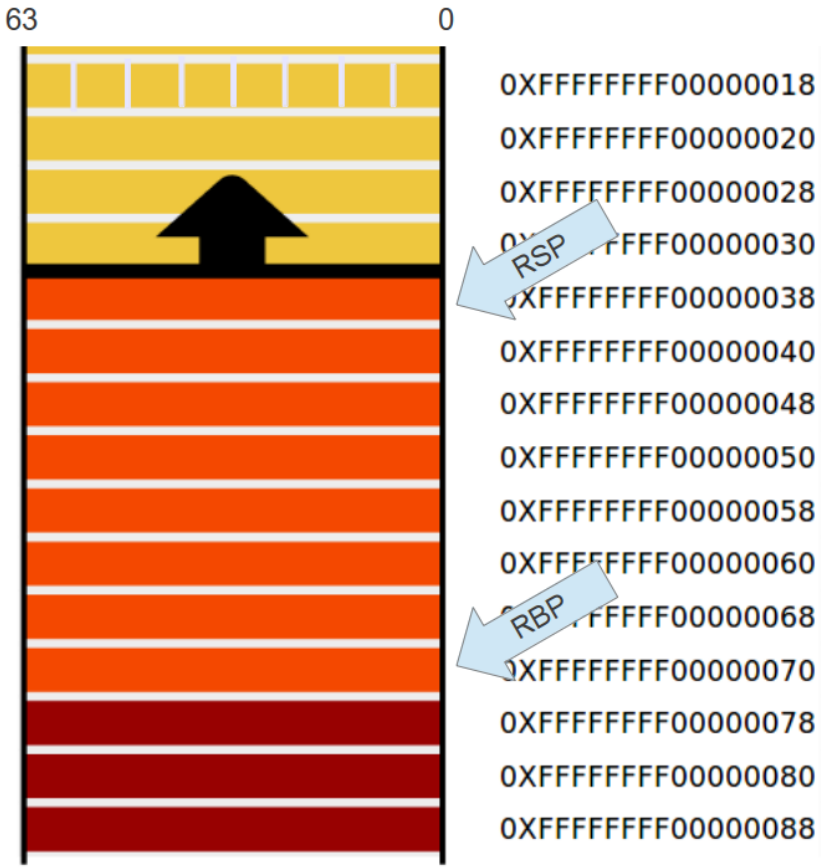
19 de agosto de 2014

# Hoy vamos a ver

- Pila
- Convención C
- Interacción C-ASM
- Ejercicios

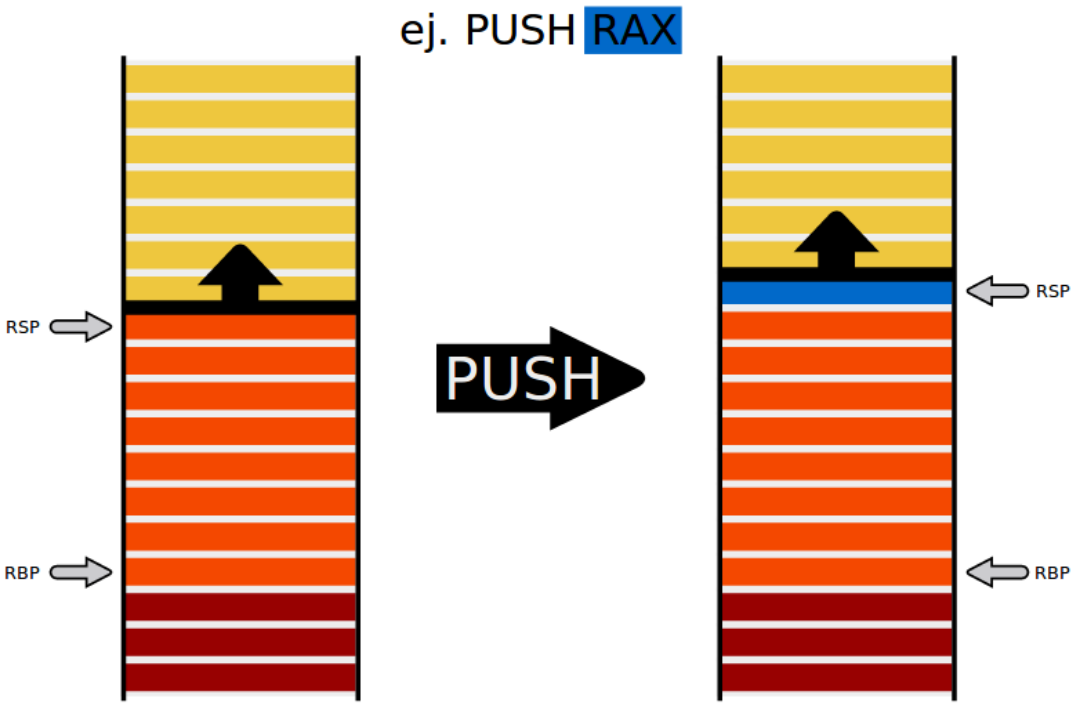
# Pila

Para ponernos de acuerdo...




- Está en memoria.
- RSP y RBP la definen.
- Crece númericamente “para atrás”.

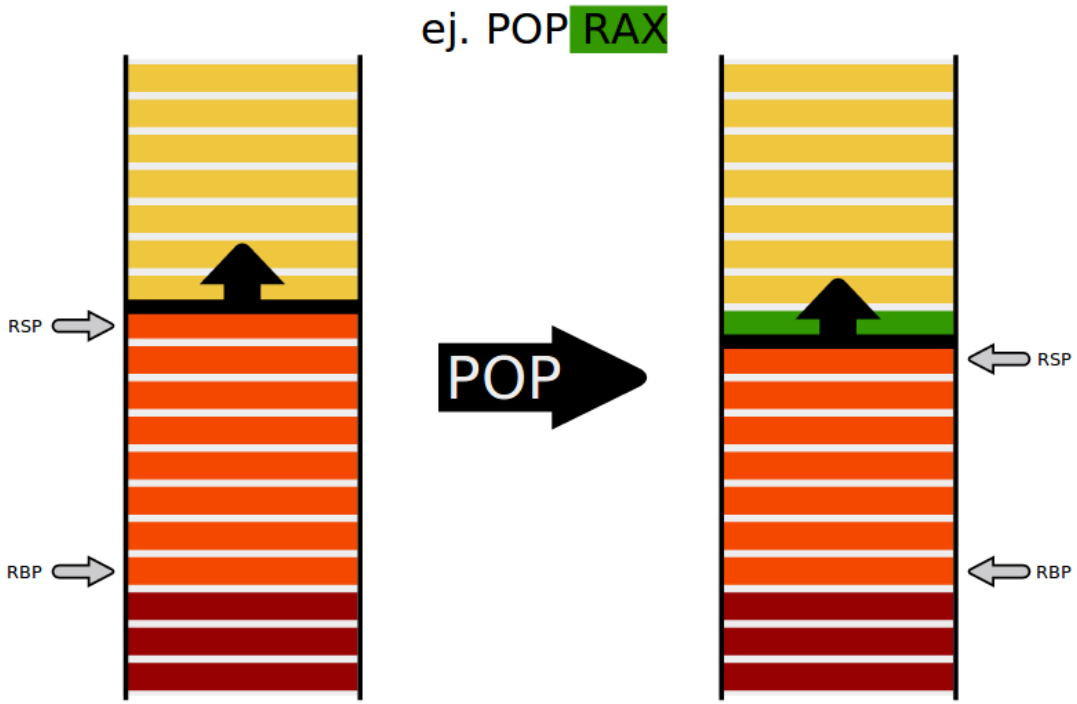
# ¿Cómo la usamos? PUSH y POP




push rax

```
sub rsp, 8
mov [rsp], rax
```

# ¿Cómo la usamos? PUSH y POP




```
pop rax
```

```
mov rax, [rsp]
add rsp, 8
```

# Stack frame

Espacio asignado en la pila cuando un proceso es llamado y removido cuando éste termina.

El bloque de información guardado en la pila para efectivizar el llamado y la vuelta, es el **stack frame**. En general el stack frame para un proceso contiene toda la información necesaria para resguardar el estado del proceso.

# Garantías en 64 bits

Una función para cumplir con la Convención C debe:

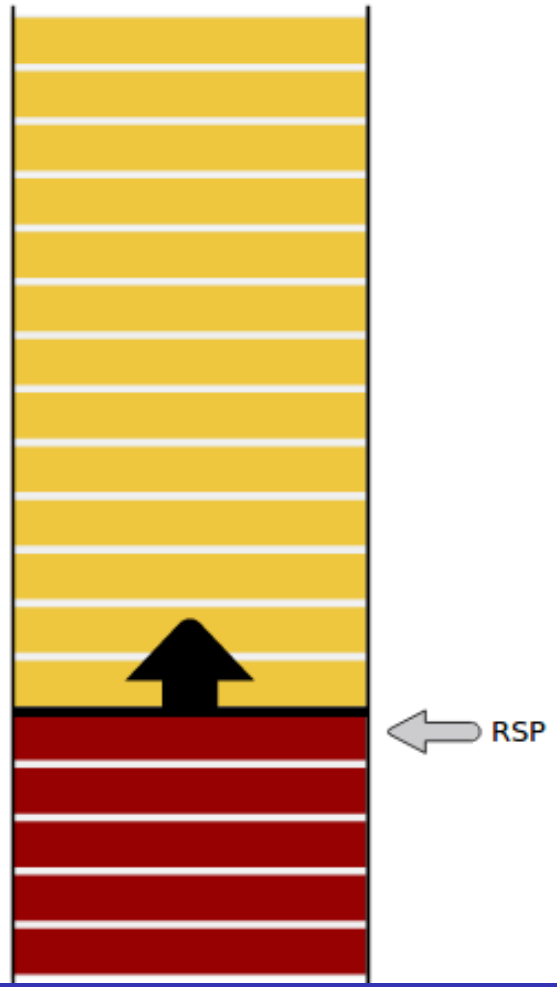
- Preservar RBX, R12, R13, R14 y R15
- Retornar el resultado en RAX o XMM0
- No romper la pila

Sólo eso, por lo cual, todo registro que no esté en la lista anterior, puede ser **modificado** por la función.

Antes de hacer un llamado, tenemos que tener la pila alineada a **16 bytes.** La alineación de la pila esta definida por RSP.
Sino...

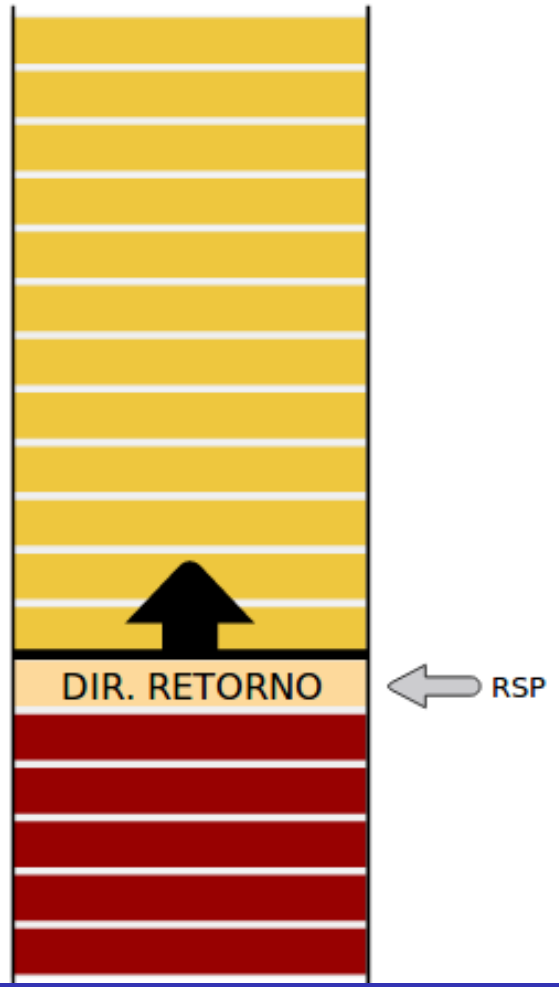
# Stack Frame - 64 bits (1)

```
fun:
 PUSH RBP
 MOV RBP,RSP
 PUSH RBX
 PUSH R12
 PUSH R13
 PUSH R14
 PUSH R15

      MI
   CÓDIGO

POP R15
 POP R14
 POP R13
 POP R12
 POP RBX
 POP RBP
 RET
```



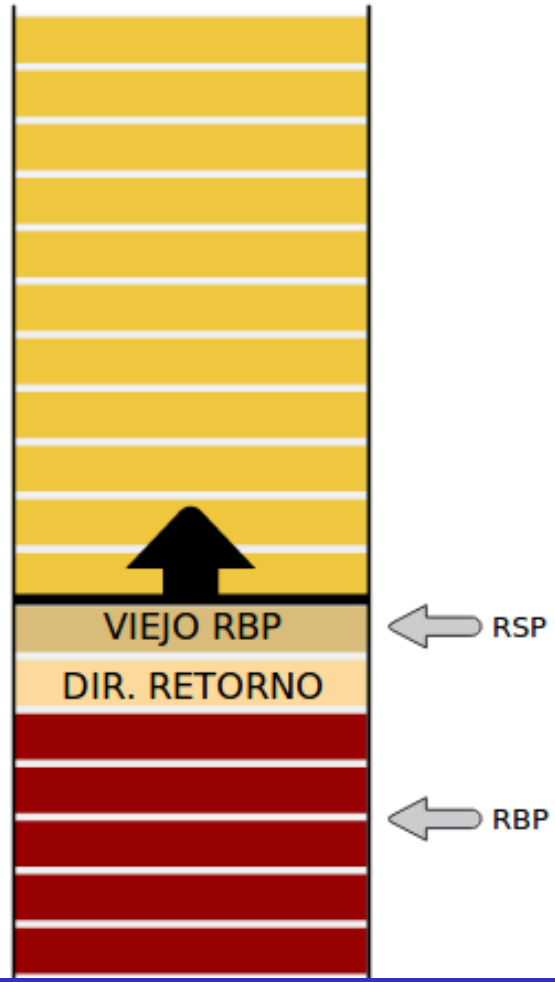
# Stack Frame - 64 bits (2)

```
fun:
 PUSH RBP
 MOV RBP,RSP
 PUSH RBX
 PUSH R12
 PUSH R13
 PUSH R14
 PUSH R15

      MI
   CÓDIGO

POP R15
 POP R14
 POP R13
 POP R12
 POP RBX
 POP RBP
 RET
```



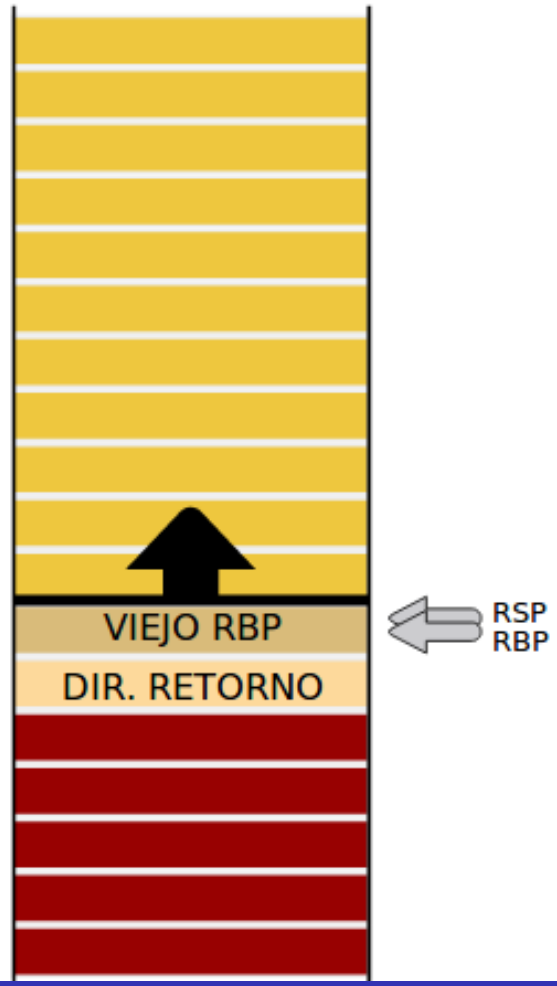
# Stack Frame - 64 bits (3)

```
fun:
 PUSH RBP
 MOV RBP,RSP
 PUSH RBX
 PUSH R12
 PUSH R13
 PUSH R14
 PUSH R15

      MI
   CÓDIGO

POP R15
 POP R14
 POP R13
 POP R12
 POP RBX
 POP RBP
 RET
```



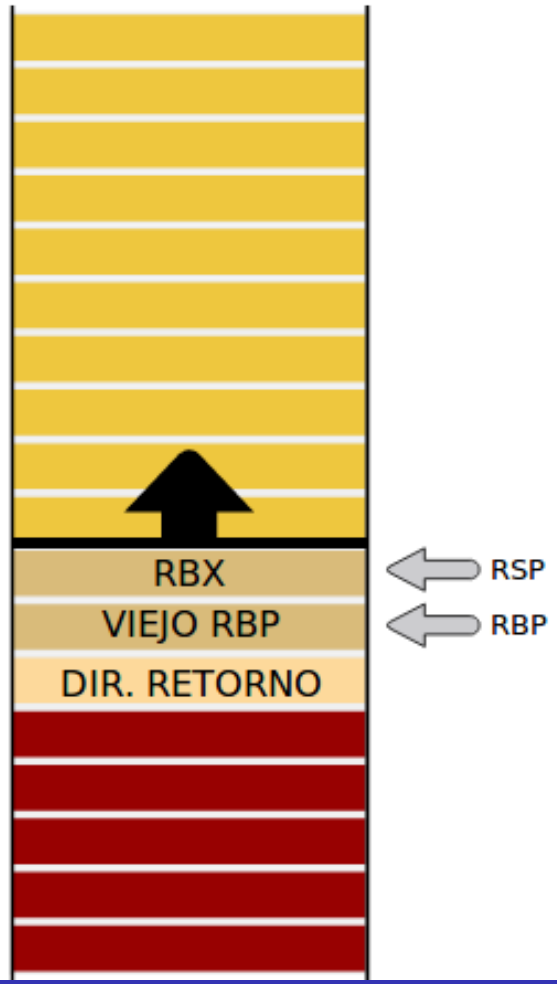
# Stack Frame - 64 bits (4)

```
fun:
 PUSH RBP
 MOV RBP,RSP
 PUSH RBX
 PUSH R12
 PUSH R13
 PUSH R14
 PUSH R15

      MI
   CÓDIGO

POP R15
 POP R14
 POP R13
 POP R12
 POP RBX
 POP RBP
 RET
```



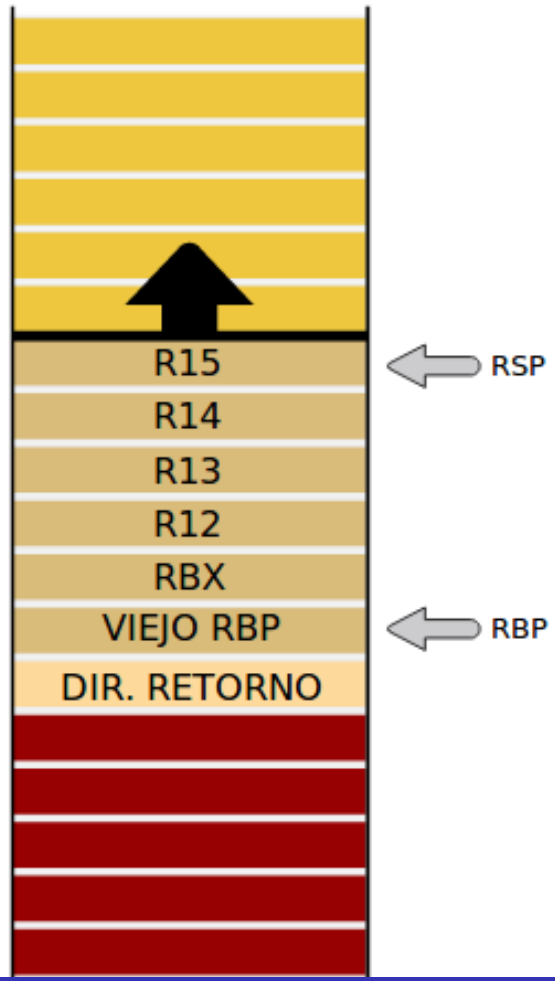
# Stack Frame - 64 bits (5)

```
fun:
 PUSH RBP
 MOV RBP,RSP
 PUSH RBX
 PUSH R12
 PUSH R13
 PUSH R14
 PUSH R15

      MI
   CÓDIGO

POP R15
 POP R14
 POP R13
 POP R12
 POP RBX
 POP RBP
 RET
```



# Stack Frame - 64 bits (6)

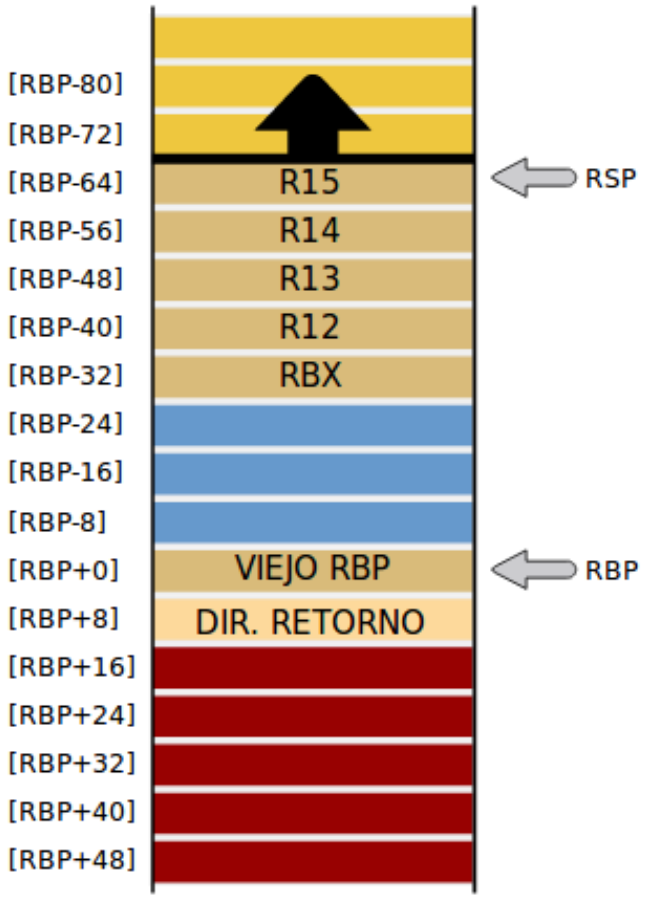
```
fun:
 PUSH RBP
 MOV RBP,RSP
 PUSH RBX
 PUSH R12
 PUSH R13
 PUSH R14
 PUSH R15
 → MI CÓDIGO
POP R15
 POP R14
 POP R13
 POP R12
 POP RBX
 POP RBP
 RET
```



# Stack Frame - 64 bits (7)

## Variables locales

```
fun:
 PUSH RBP
 MOV RBP,RSP
 SUB RSP,24
 PUSH RBX
 PUSH R12
 PUSH R13
 PUSH R14
 PUSH R15

      MI
   CÓDIGO

POP R15
 POP R14
 POP R13
 POP R12
 POP RBX
 ADD RSP,24
 POP RBP
 RET
```



# Garantías en 32 bits

Una función para cumplir con la Convención C debe:

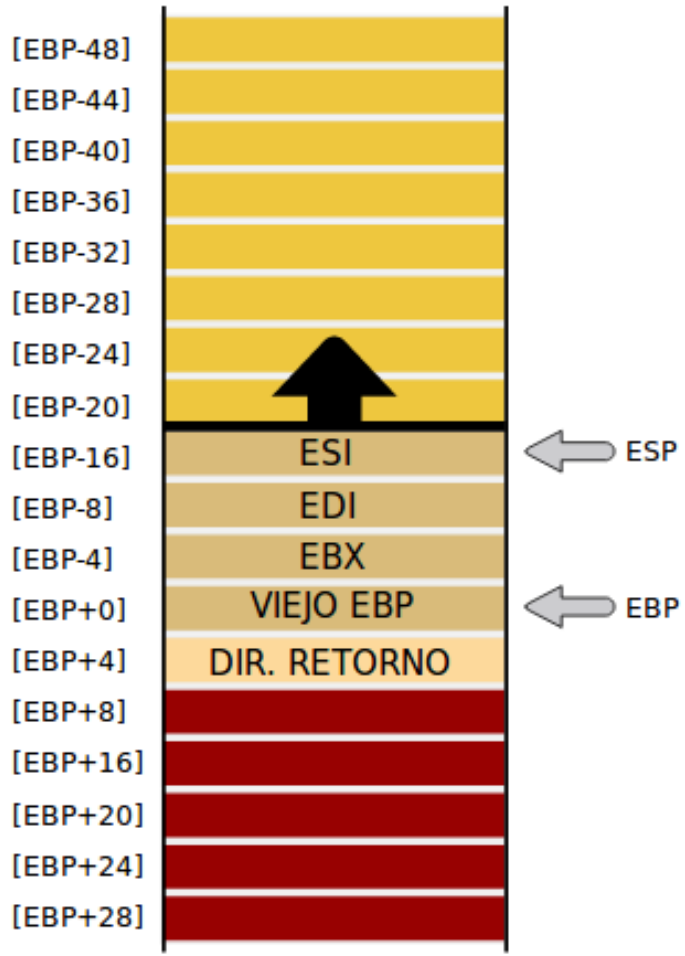
- Preservar EBX, ESI Y EDI
- Retornar el resultado en EAX
- No romper la pila

Solo eso, por lo cual, todo registro que no esté en la lista anterior, puede ser **modificado** por la función.

Antes de hacer un llamado, tenemos que tener la pila alineada a **4 Bytes**

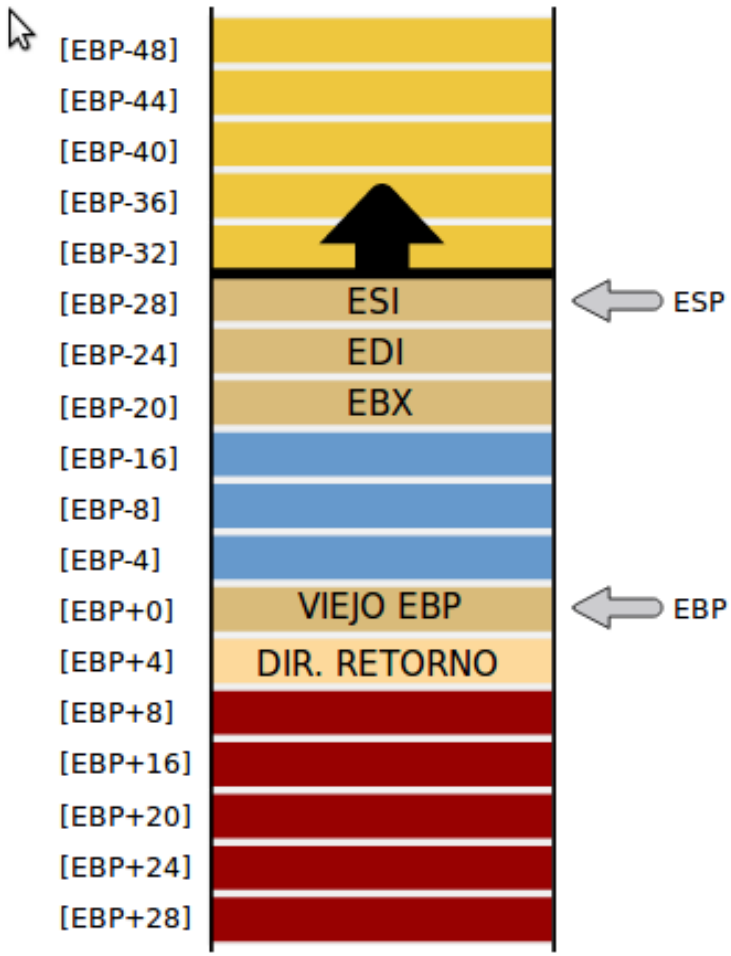
# Stack Frame - 32 bits (1)

```
fun:
 PUSH EBP
 MOV EBP,ESP
 PUSH EBX
 PUSH EDI
 PUSH ESI
      MI
      CÓDIGO
 POP ESI
 POP EDI
 POP RBX
 POP RBP
 RET
```



# Stack Frame - 32 bits (2)

```
fun:
 PUSH EBP
 MOV EBP,ESP
 SUB ESP,12
 PUSH EBX
 PUSH EDI
 PUSH ESI

      MI
    CÓDIGO

 POP ESI
 POP EDI
 POP RBX
 ADD ESP,12
 POP RBP
 RET
```



# Pasaje de parámetros

## En 64 bits

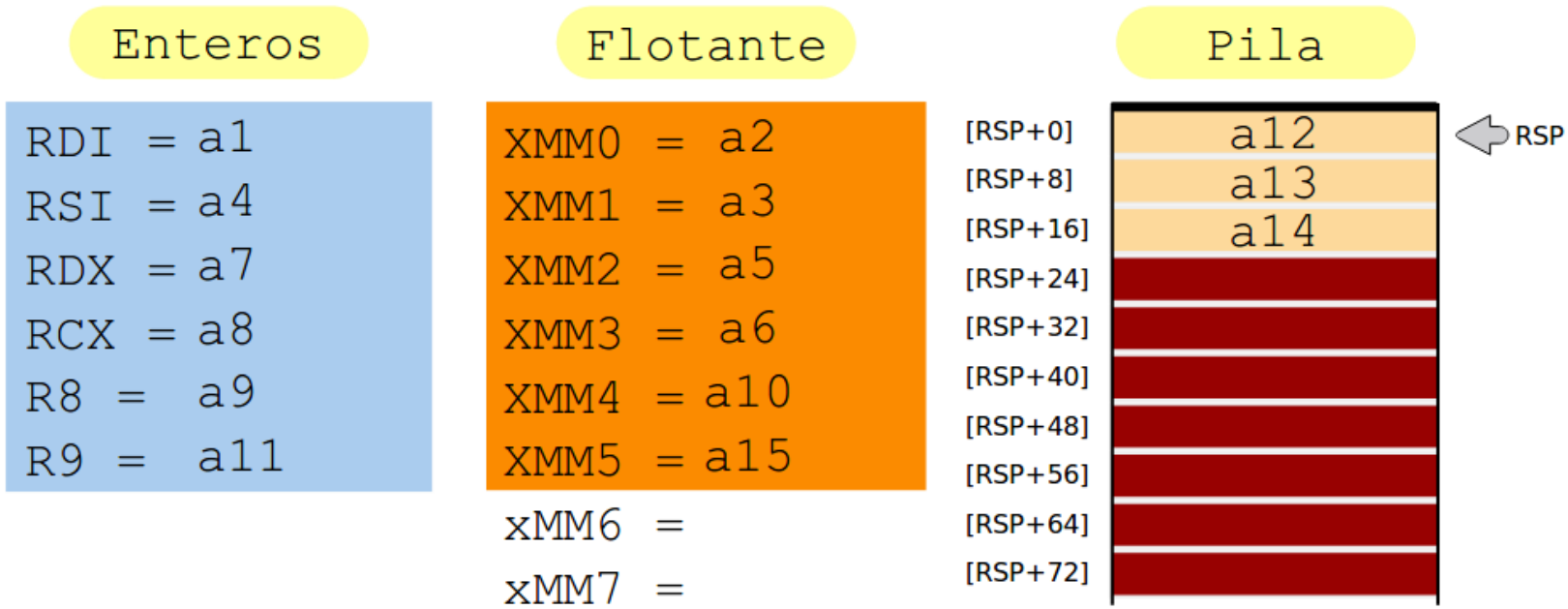
Los parámetros se pasan (de izquierda a derecha) por los registros

- Si es **entero** o **puntero** se pasan respetando el orden usando:
  - `RDI`, `RSI`, `RDX`, `RCX`, `R8` y `R9`
- Si es de tipo **flotante** se pasan en los `XMMs`

Si no hay más registros disponibles se usa la pila, pero deberán quedar ordenados desde la dirección más baja a la más alta (se pushean de derecha a izquierda)

# Pasaje de parámetros - Ejemplo en 64 bits

```
int f( int a1, float a2, double a3, int a4, float a5,
 double a6, int* a7, double* a8, int* a9, double a10,
 int** a11, float* a12, double** a13, int* 14, float a15)
```



# Pasaje de parámetros

## En 32 bits

Los parámetros se pasan por la pila.

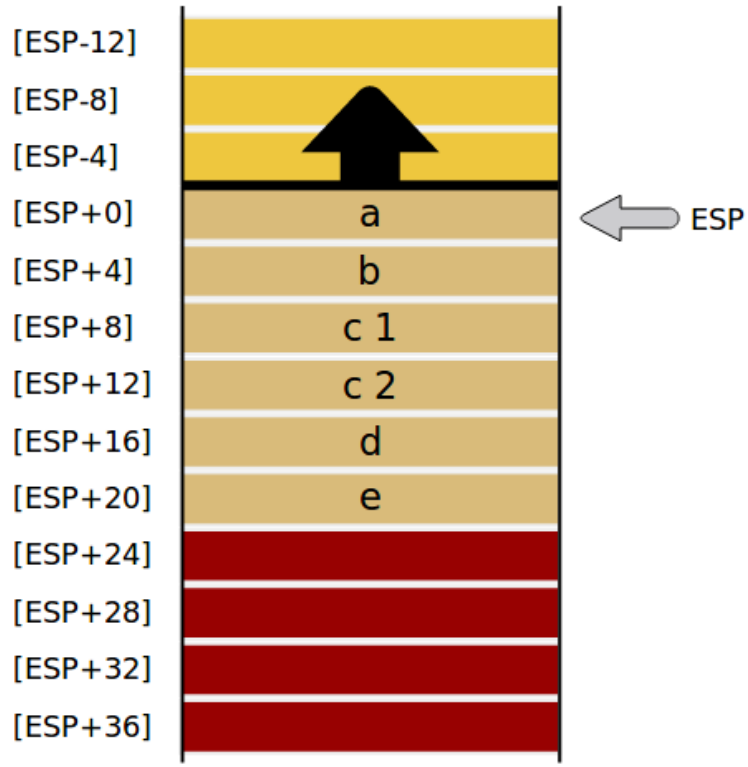
Deben quedar ordenados desde la dirección más baja a la más alta. (se pushean de derecha a izquierda)

# Pasaje de parámetros - Ejemplo en 32 bits

```
int f1( int a, float b, double c, int* d, double* e)
```

llamado...

```
...
push e
push d
push c1
push c2
push b
push a
call f1
add esp, 6*4
...
```



# Llamar funciones ASM desde C

## funcion.asm

```
global fun
section .text
fun:
...
...
ret
```

## programa.c

```
extern int fun(int, int);
int main(){
    ...
    fun(44,3);
    ..
}
```

## ensamblar, compilar y linkear

```
nasm -f elf64 funcion.asm -o funcion.o
gcc -o ejec programa.c funcion.o
```

# Llamar funciones C desde ASM

## main.asm

```
global main
extern fun
section .text
main:
    ...
    call fun
    ...
    ret
```

## funcion.c

```
int fun(int a, int b){
    ...
    ...
    int res= a+b;
    ...
    return res;

}
```

## ensamblar, compilar y linkear

```
nasm -f elf64 main.asm -o main.o
gcc -c -m64 funcion.c -o funcion.o
gcc -o ejec -m64 main.o funcion.o
```

# Ejercicios

1. Armar un programa en `C` que llame a una función en `ASM` que sume dos enteros. La de `C` debe imprimir el resultado.

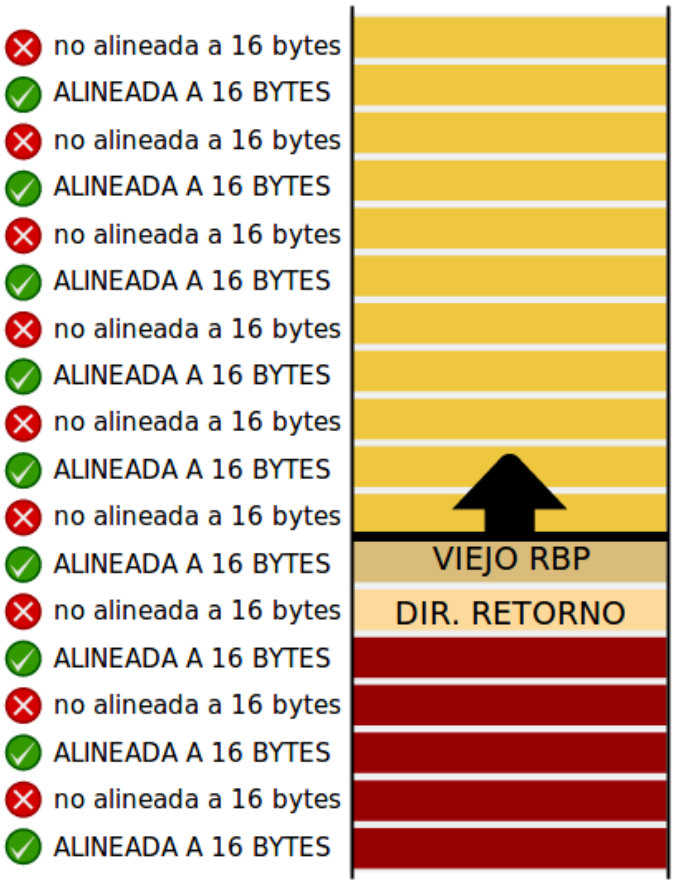
2. Modificar la función anterior para que sume dos numeros de tipo `double`. (ver `ADDPD`)

3. Construir una función en `ASM` que imprima correctamente por pantalla sus parámetros en orden, llamando sólo una vez a `printf`. La función debe tener la siguiente aridad:
```
void imprime_parametros( int a, double f, char* s );
```

4. Construir una función en `ASM` con la siguiente aridad:
```
int suma_parametros( int a0, int a1, int a2, int a3,
                     int a4, int a5 ,int a6, int a7 );
```
La función retorna como resultado la operación:
`a0-a1+a2-a3+a4-a5+a6-a7`

# Notas de clase

Recuerden que para hacer cualquier llamada a una función desde ASM tienen que tener la pila alineada.




(3)
(2)
(1)

(1) - Inicialmente la pila esta alineada a 16 bytes

(2) - Cuando se hace un CALL se guarda la dirección de retorno y se desalinea

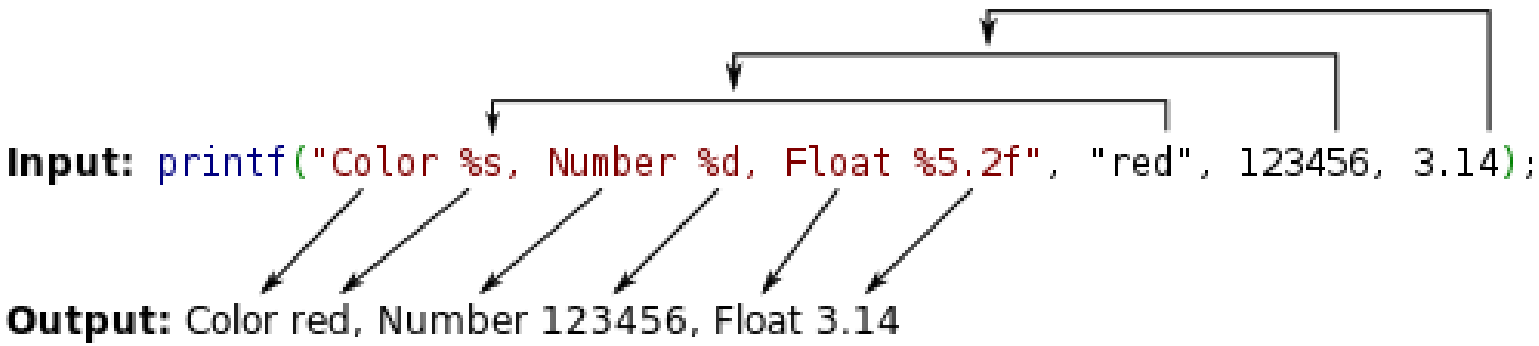
(3) - Cuando armamos el StackFrame guardamos el viejo RBP y alineamos la pila a 16 bytes

(3bis) - Otra opción es restar al RSP 8 bytes para alinear la pila. Es una mala practica usar la instrución Push para hacer esto.

La desición de armar el StackFrame depende del programador, se recomienda armarlo si se va a ser uso de variables locales o parámetros en la pila.

# Notas de clase

Para imprimir por pantalla vamos a usar printf.




Importante: Si se la llama desde `ASM`, entonces el contenido de `RAX` tiene que estar en 1.

[*] http://www.cplusplus.com/reference/clibrary/cstdio/printf/

Manuales de intel

- `orga2.exp.dc.uba.ar` > *Recursos* > *Manuales de Intel*.
- `http://orga2.exp.dc.uba.ar/index.php?pid=27`