Pila
Convención C
Interacción C-ASM

Organización del Computador II

20 de marzo de 2018

Hoy vamos a ver

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

Por qué vamos a verlo

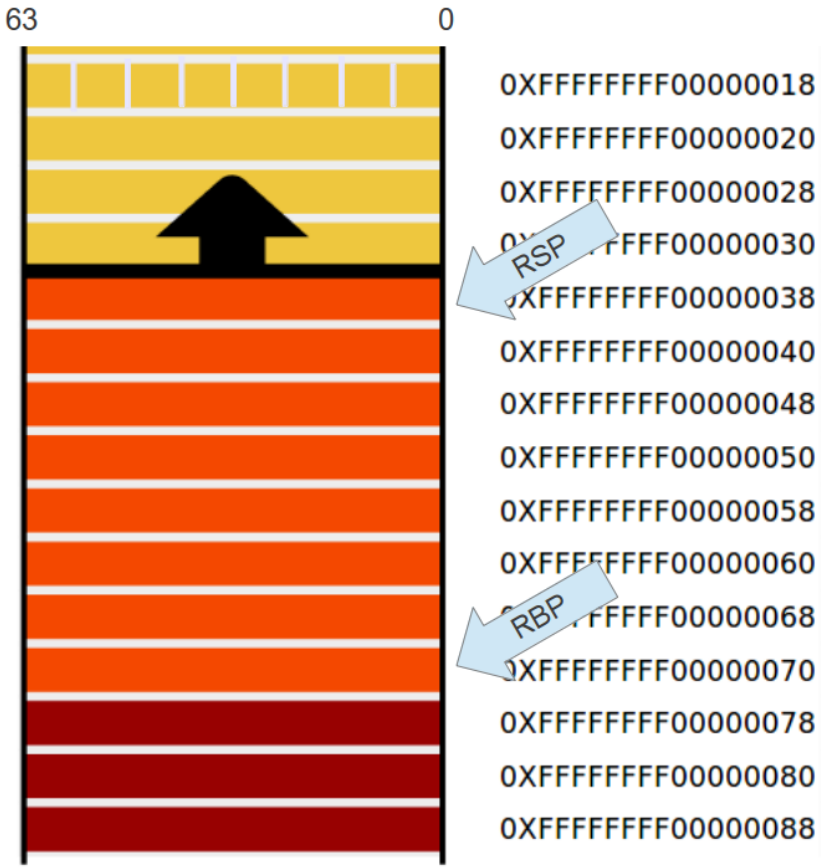
- La pila es el mecanismo de almacenamiento e intercambio entre contextos más usado luego de los registros
- Necesitamos/queremos interactuar con código escrito en C
- Es un primer caso de uso de políticas de uso que debemos respetar

Donde nos vamos a equivocar

- No vamos a entender el por qué de la pila
- No vamos a alinear la pila
- Vamos a romper la convención C
- Vamos a esperar que la convención C preserve más cosas de las que debe

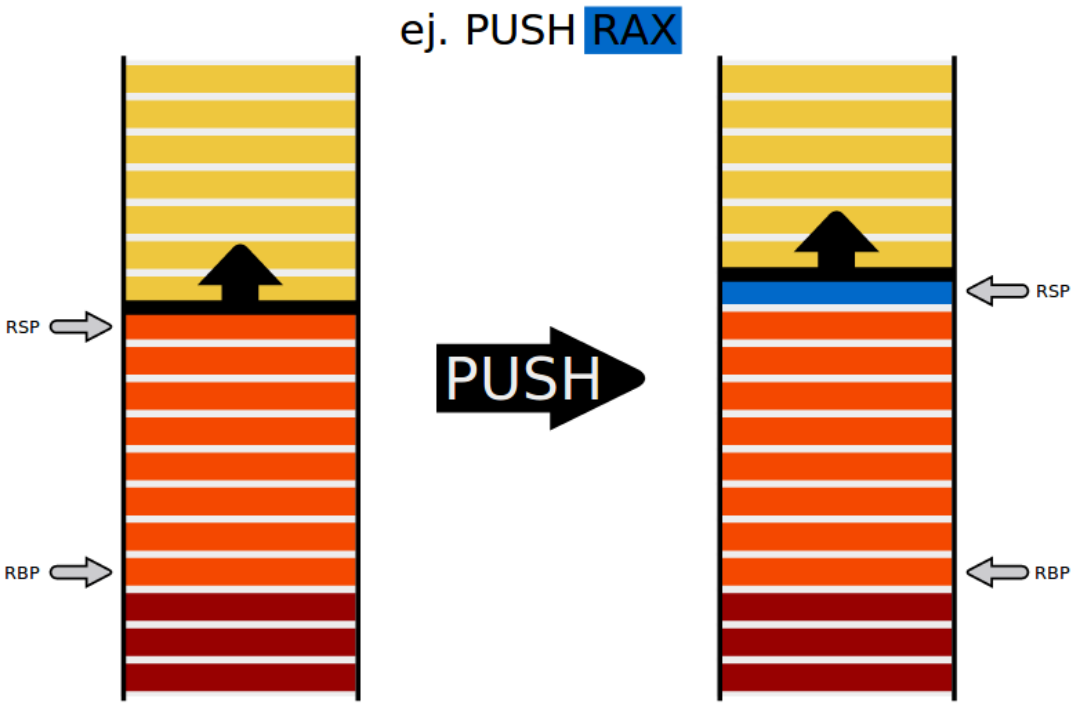
Pila

Para ponernos de acuerdo...

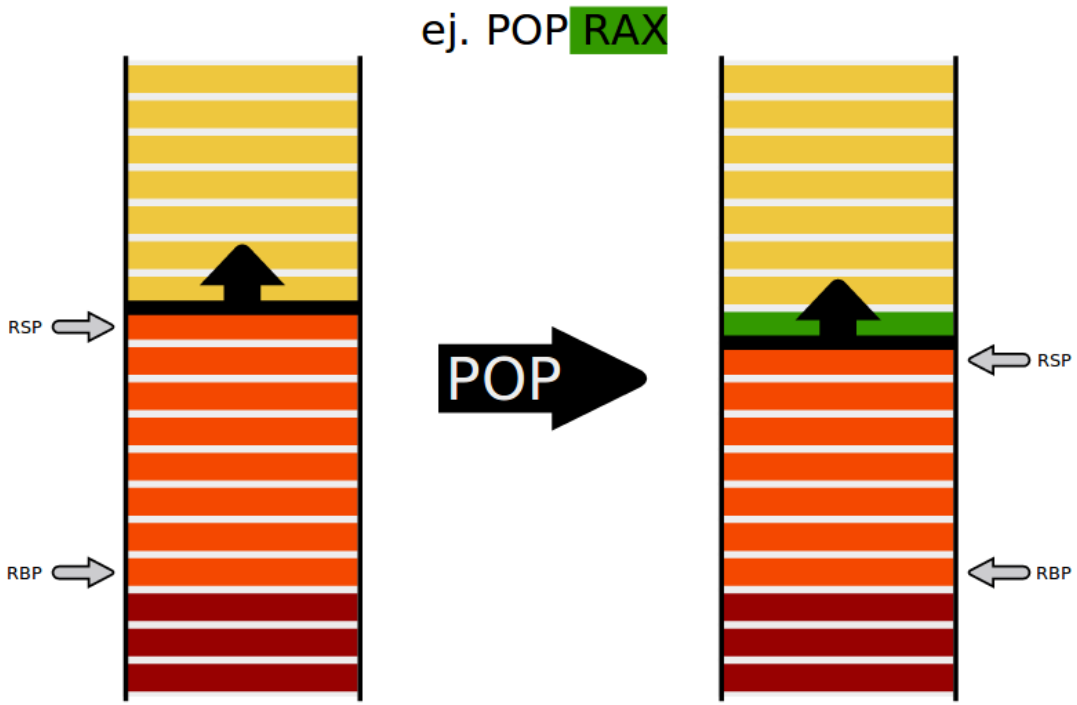


- Está en memoria.
- RSP y RBP la definen.
- Crece númericamente “para atrás”.

¿Cómo la usamos? PUSH y POP



¿Cómo la usamos? PUSH y POP



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.

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.

Nota: La pila está alineada a **8 bytes**. Pero cuando interactuamos con funciones de C en 64 bits, antes de hacer un llamado a una de ellas, tenemos que tener la pila alineada a **16 bytes.** La alineación de la pila está definida por RSP. Si no...

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.

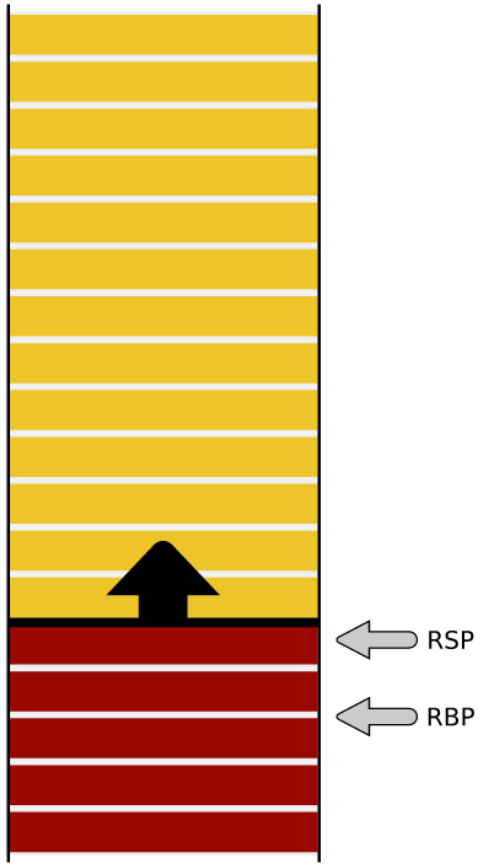
Nota: La pila está alineada a **8 bytes**. Pero cuando interactuamos con funciones de C en 64 bits, antes de hacer un llamado a una de ellas, tenemos que tener la pila alineada a **16 bytes.** La alineación de la pila está definida por RSP. Si no...

Garantías en 64 bits

Esto delata por primera vez la existencia de contratos de bajo nivel cuyo motivo comprenderemos (con suerte) más adelante. Hablando de contratos pasemos a las reglas implícitas a la hora de armar y desarmar el `stack frame`.

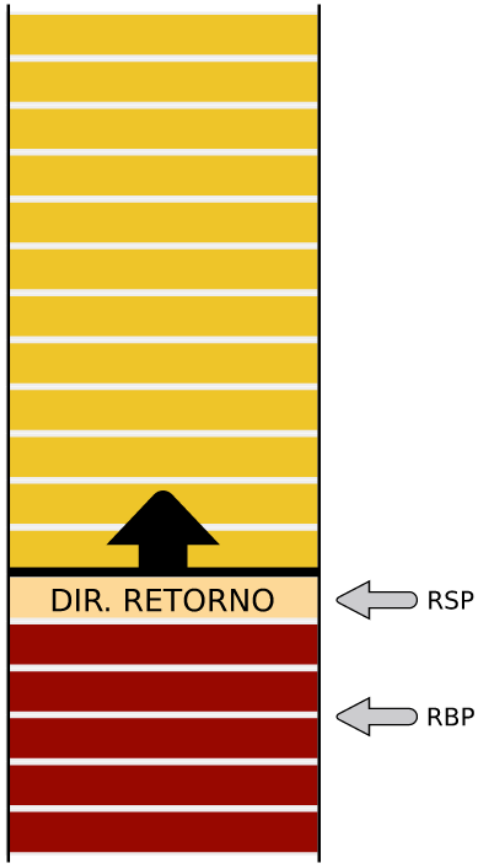
Stack Frame - 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... mas codigo ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



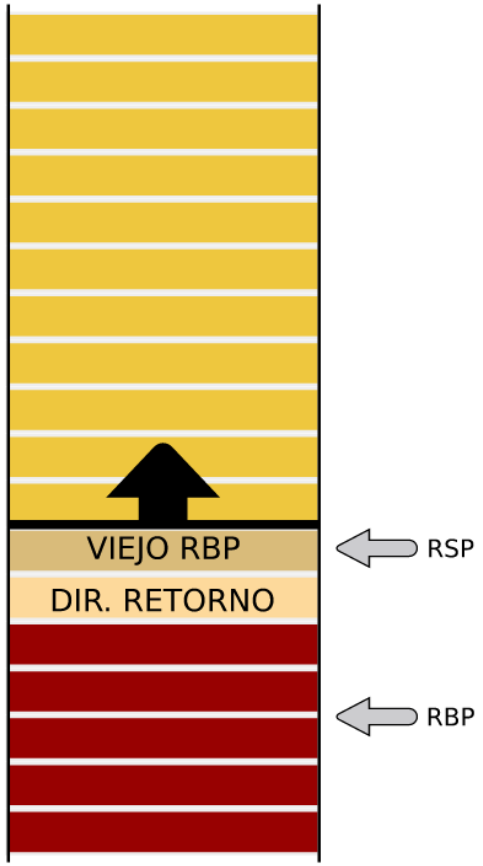
Stack Frame - 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... mas codigo ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



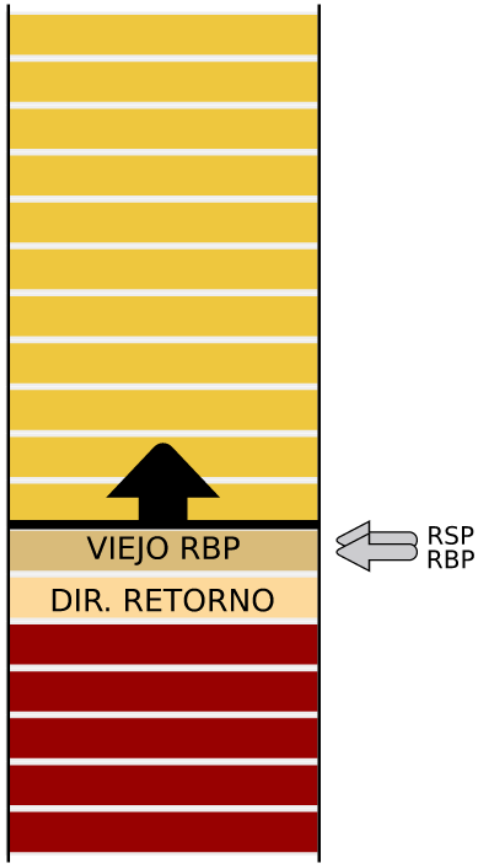
Stack Frame - 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... mas codigo ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



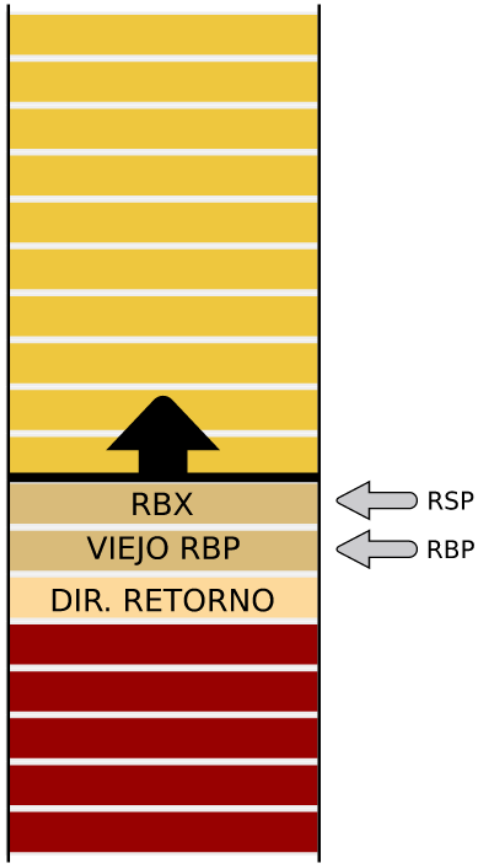
Stack Frame - 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... mas codigo ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



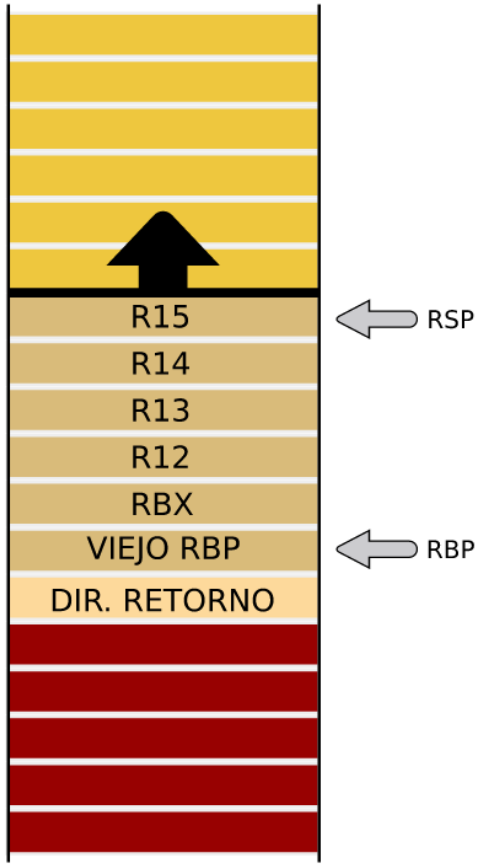
Stack Frame - 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... mas codigo ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



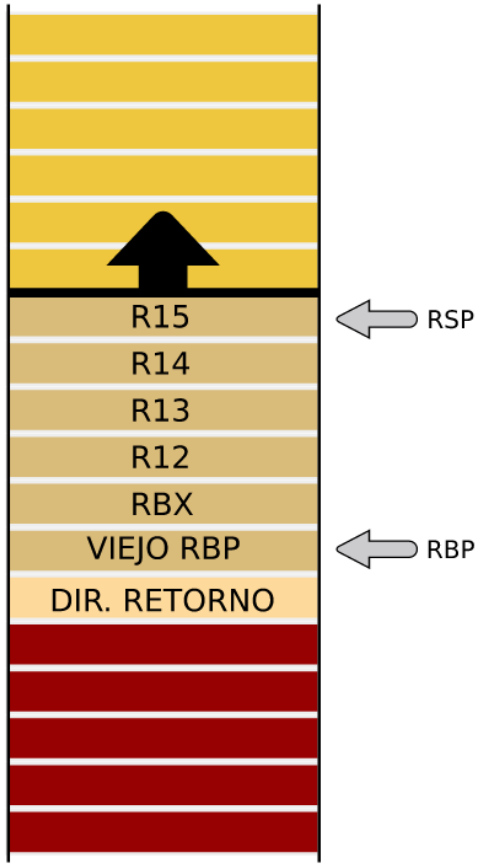
Stack Frame - 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... mas codigo ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



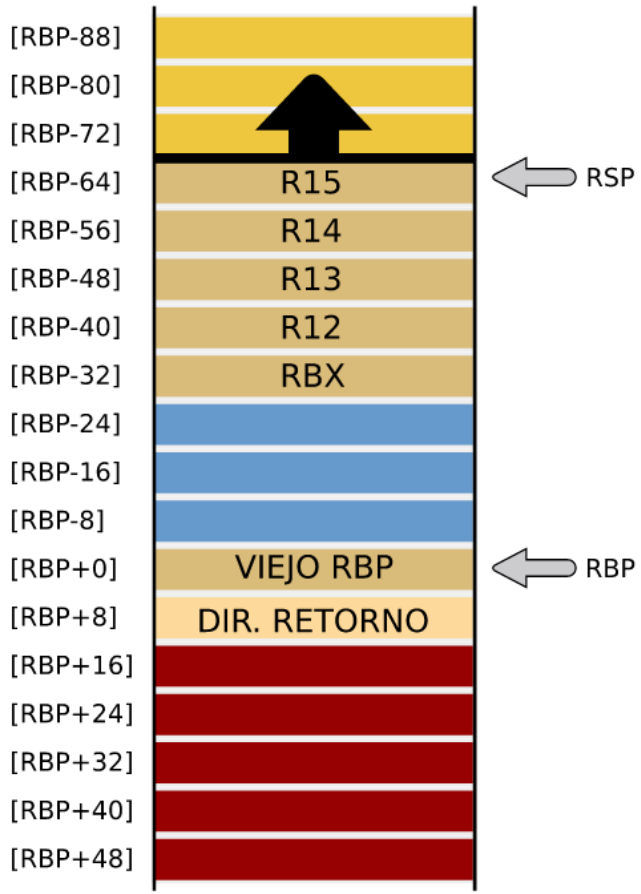
Stack Frame - 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... mas codigo ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



Stack Frame - 64 bits

```
fun:
   push rbp
   mov rbp, rsp
   sub rsp, 24
   push rbx
   push r12
   push r13
   push r14
   push r15
   ... mas codigo ...
   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:

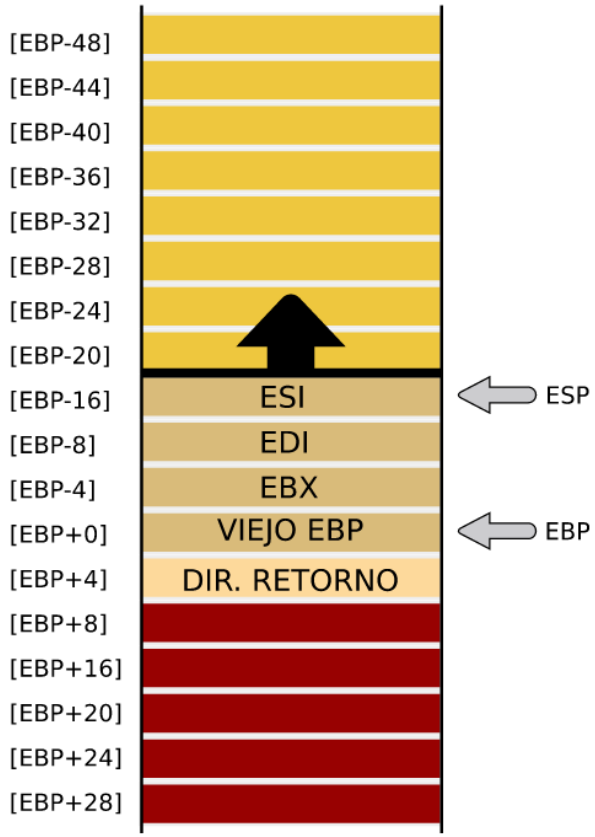
- Preservar EBX, ESI Y EDI
- Retornar el resultado en EAX
- No romper la pila

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

Nota: La pila está alineada a **4 bytes**. Antes de hacer un llamado a una función tenemos que tener la pila alineada a **4 bytes**.

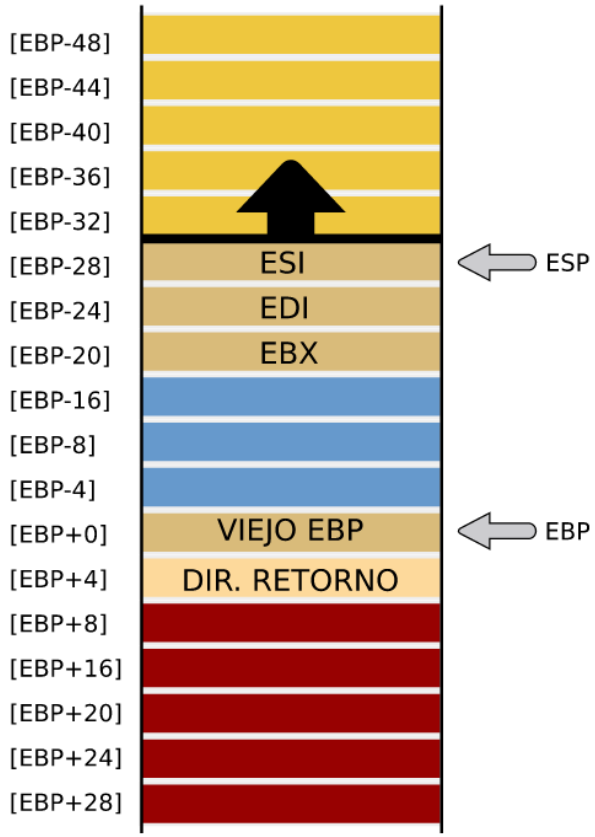
Stack Frame - 32 bits

```
fun:
  push ebp
  mov ebp, esp

  push ebx
  push edi
  push esi
  ... mas codigo ...
  pop esi
  pop edi
  pop ebx

  pop ebp
  ret
```



Stack Frame - 32 bits

```
fun:
   push ebp
   mov ebp, esp
   sub esp, 12
   push ebx
   push edi
   push esi
   ... mas codigo ...
   pop esi
   pop edi
   pop ebx
   add esp, 12
   pop ebp
   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)

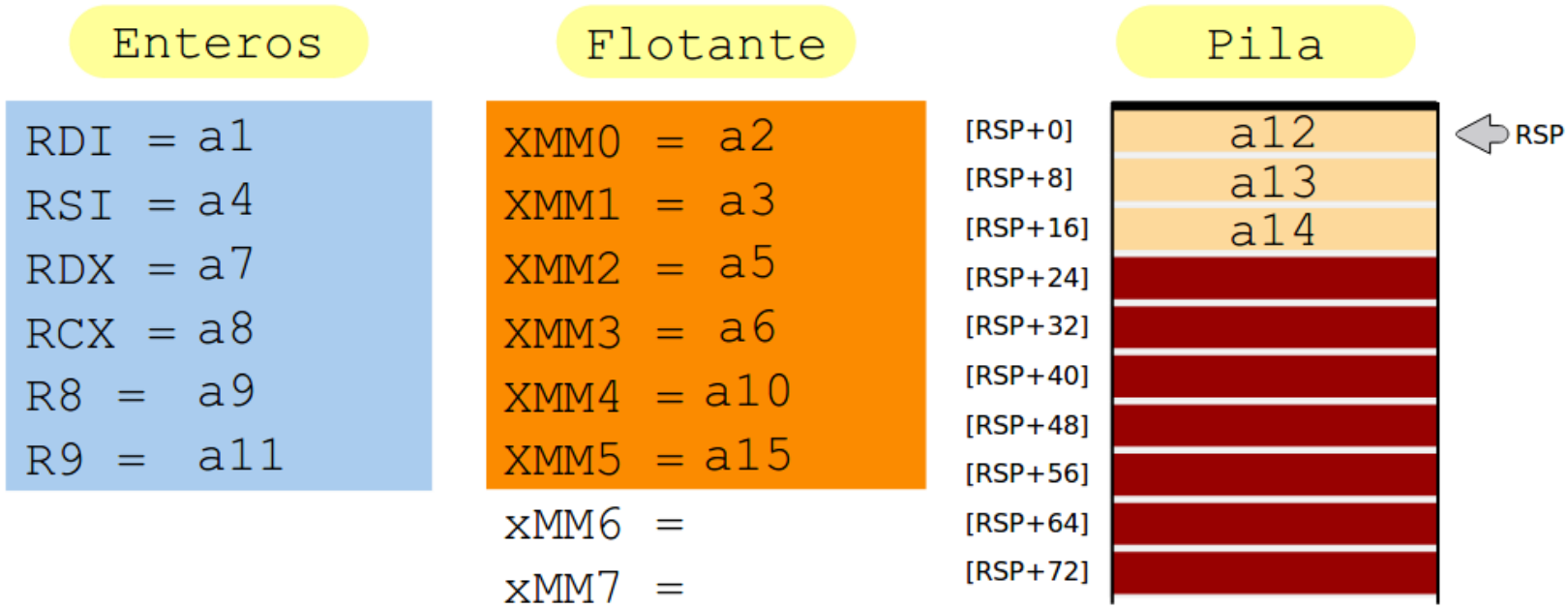
Pasaje de parámetros - Ejemplo en 64 bits

```
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* a14, float a15)
```

Enteros	Flotante	Pila
RDI =	XMM0 =	
RSI =	XMM1 =	
RDX =	XMM2 =	
RCX =	XMM3 =	
R8 =	XMM4 =	
R9 =	XMM5 =	
	xMM6 =	
	xMM7 =	

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* a14, 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

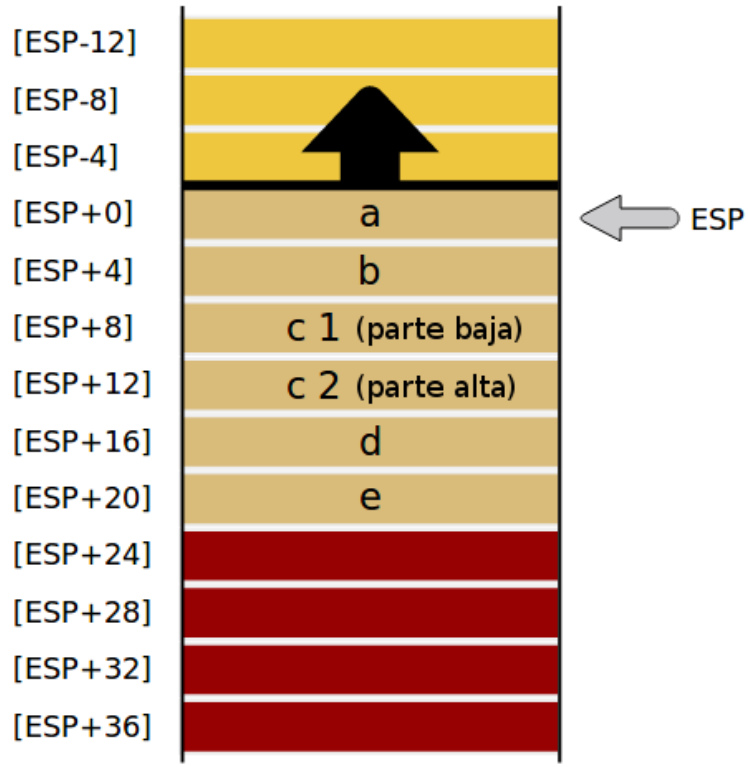
```
int f1( int a, float b, double c, int* d, double* e)
```

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 c2
push c1
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.

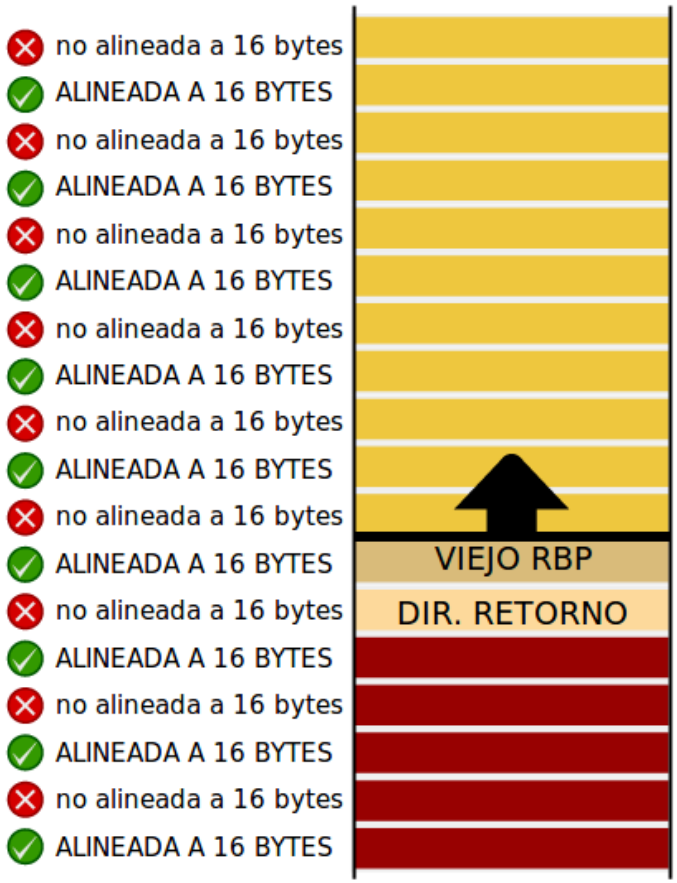
2. Modificar la función anterior para que sume dos numeros de tipo `double`. (ver `ADDPD`)

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.



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

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

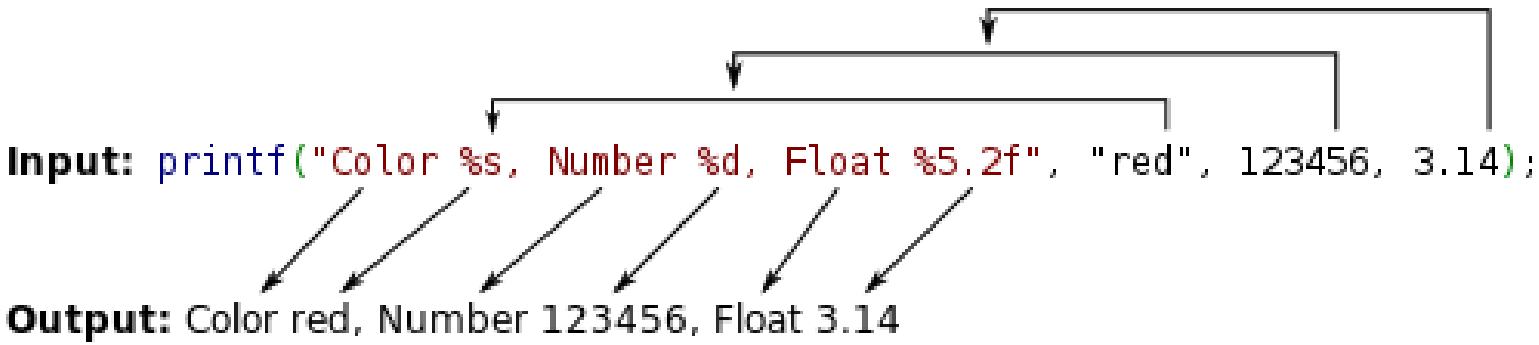
(3) - Cuando armamos el StackFrame guardamos el viejo RBP y alineamos la pila a 16 bytes

(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

- `http://www.dc.uba.ar/materias/oc2/2017/c1` > *Descargas* > *Manuales de Intel*.
- `http://www.dc.uba.ar/materias/oc2/2017/c1/Descargas`