Pila

Convención C / Interacción C-ASM

Organización del Computador II - 1er Cuatrimestre 2019

Departamento de Computación - FCEyN - UBA

19 de Marzo de 2019

Temario

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

Introducción

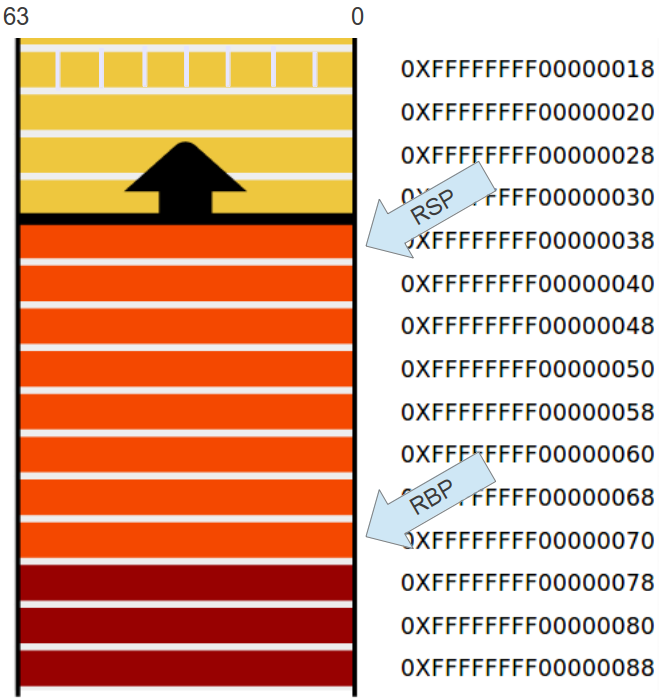
- La pila es una estructura en memoria para almacenar la información de diferentes contextos
- Nos permite guardarla y restaurarla al pasar de uno a otro
- Suele usarse en los llamados a funciones, interrupciones y otros escenarios que ya veremos
- Hoy la usaremos para interactuar desde ASM con código escrito en C

Primeros pasos

- Con el correr de las clases entenderemos mejor su uso
- Debemos respetar una serie de políticas para que funcione correctamente
- Al principio podrán surgir errores de alineación y la convención C (que veremos en breve)
- Es importante atender como puede alterarse su contenido por cada cosa que vamos haciendo

Estructura

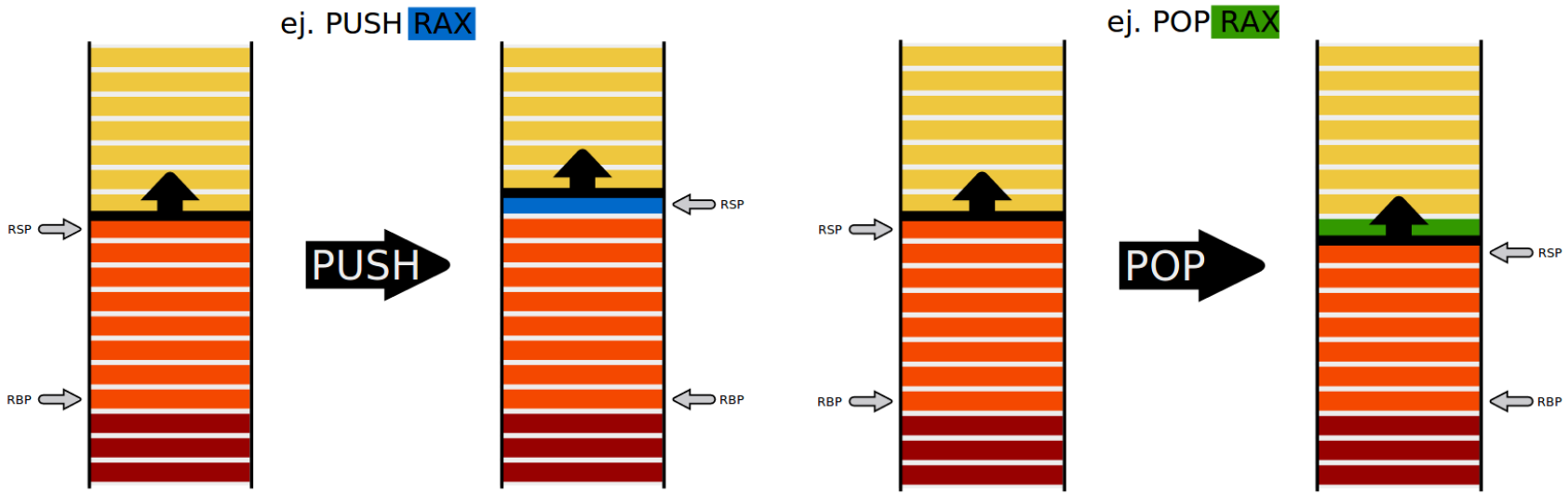
Esquemáticamente la pila se representa:



- La referenciamos con los registros RBP y RSP (en 64b) o EBP y ESP (en 32b)
- RBP (Base Pointer) apunta a su base y RSP (Stack Pointer) a la posición en que puede apilarse el próximo elemento (tope)
- Sus direcciones crecen numéricamente hacía el origen o “al revés”

Interfaz

Interactuamos con la pila a través de las operaciones `PUSH` (apilar) y `POP` (desapilar):



Convenciones de llamada a funciones

- Al trabajar con subrutinas de C debemos atender a la convención de llamada a funciones del sistema
- Ésta nos define detalladamente cómo las funciones reciben parámetros y luego retornan su resultado
- También especifica la información a preservar
- Usaremos la convención `System V AMD64 ABI` de microprocesadores x86 de 64b, actualmente estandarizada en muchos sistema Unix

Stack frame

A cada llamado a función se le asigna un espacio en la pila que es liberado al terminar. Este bloque puede resguardar información, variables locales y parámetros de otras funciones (si los registros no alcanzan). Este es el **stack frame** y lo armaremos para cada función de nuestros programas.

Caso 64 bits

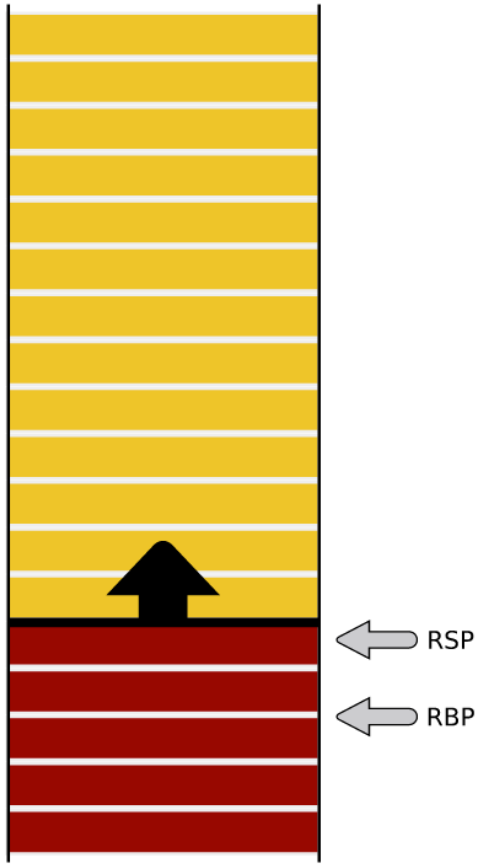
- Nuestra convención ofrece sus garantías a las funciones que:
  - Preserven los registros `RBX`, `R12`, `R13`, `R14` y `R15`
  - Retornen el resultado en `RAX` (y `RDX` si ocupa 128b) o `XMM0` (si es un número de punto flotante)
  - Preserven el estado de la pila en el retorno (preservando `RBP`)
- Ojo que hay que preservar y restaurar los registros con información valiosa que la convención no cubre
- La pila trabaja normalmente alineada a **8 bytes** (valor de `RSP`). Pero al llamar a funciones de `C` debe estarlo a **16 bytes**, sino pueden haber consecuencias **catastróficas**

Caso 64 bits (continuación)

- Respetar la convención conlleva una serie de contratos de bajo nivel
- Estos se ligaron a la arquitectura aún tras el surgimiento de nuevos sistemas operativos y compiladores
- Algunas veces su implementación se dio por cuestiones históricas (más en la sección bibliográfica)
- Veremos ahora cómo armar y desarmar el stack frame de una función genérica en ASM

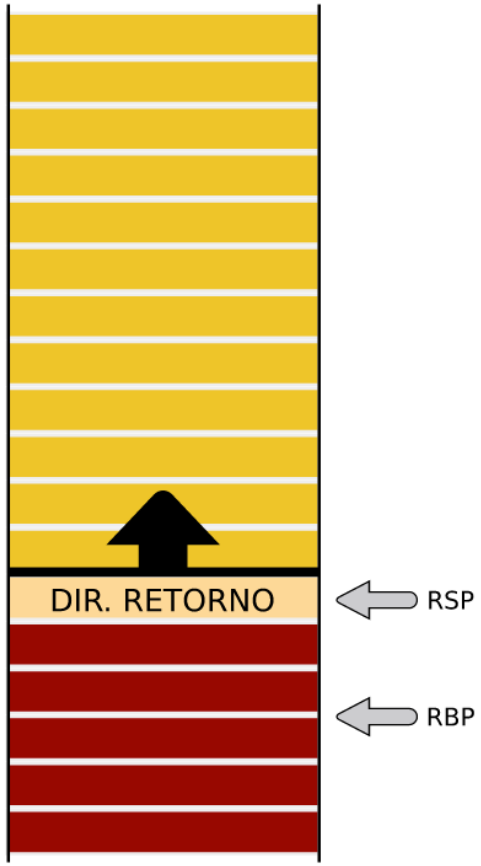
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



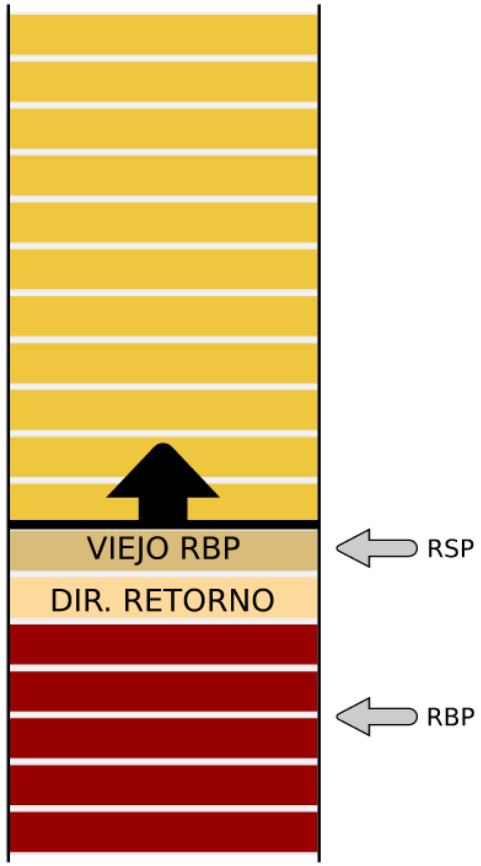
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



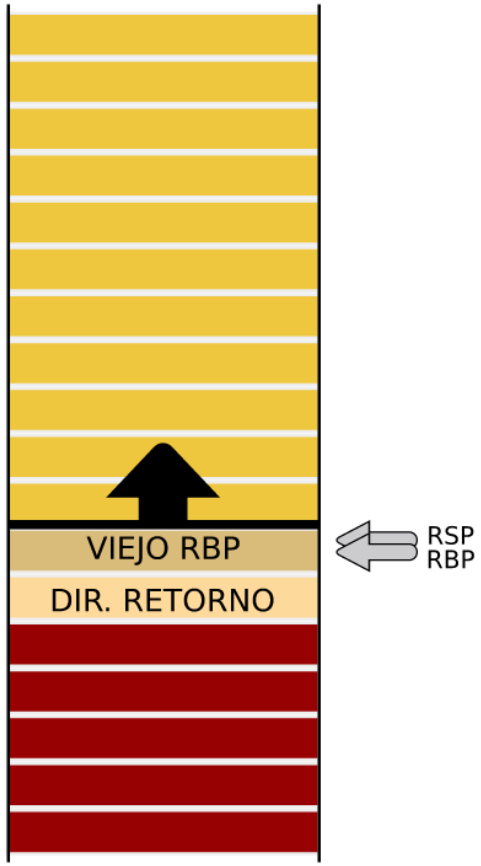
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



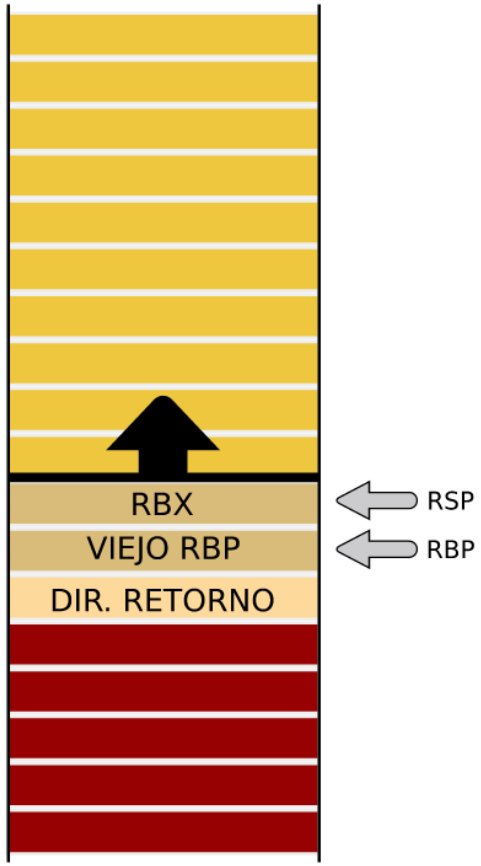
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



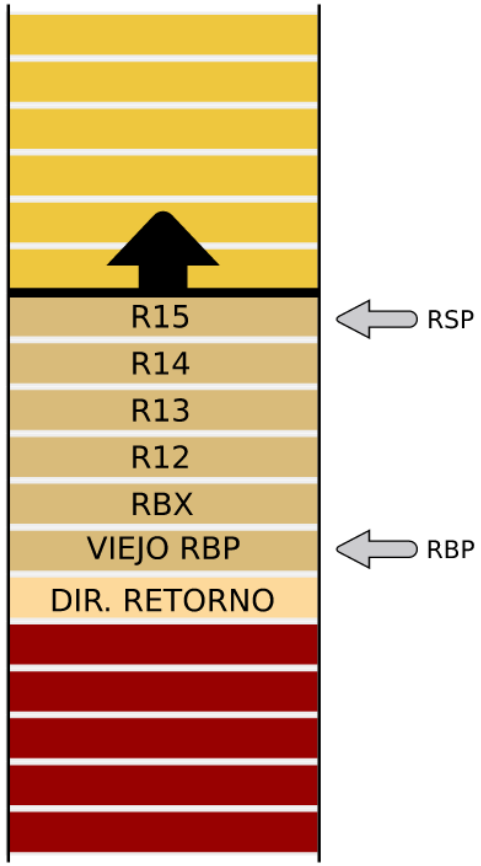
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



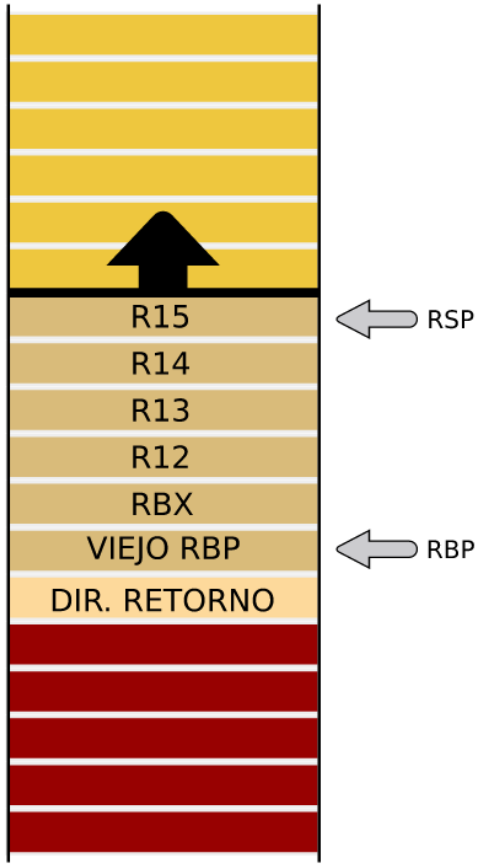
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



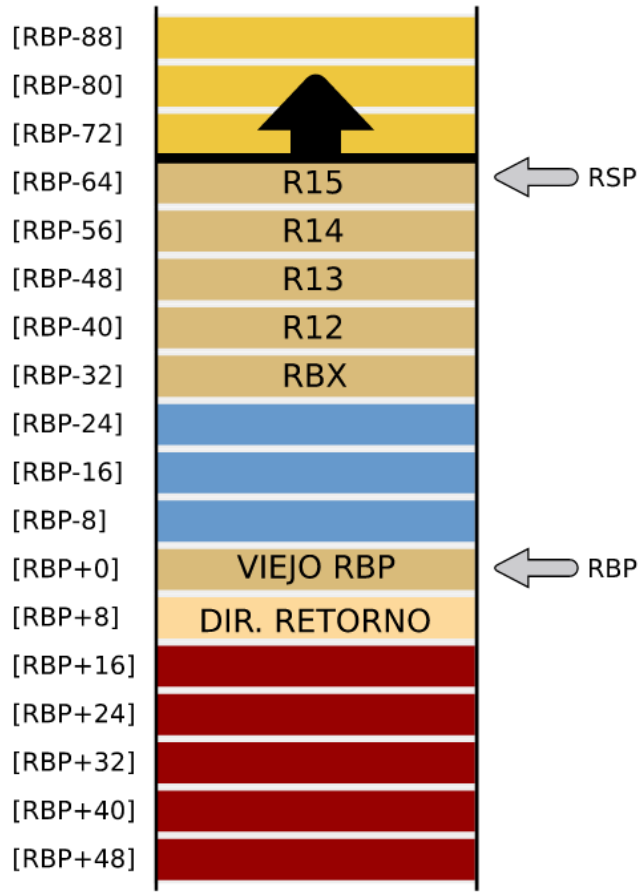
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp
   sub rsp, 24
   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx
   add rsp, 24
   pop  rbp
   ret
```

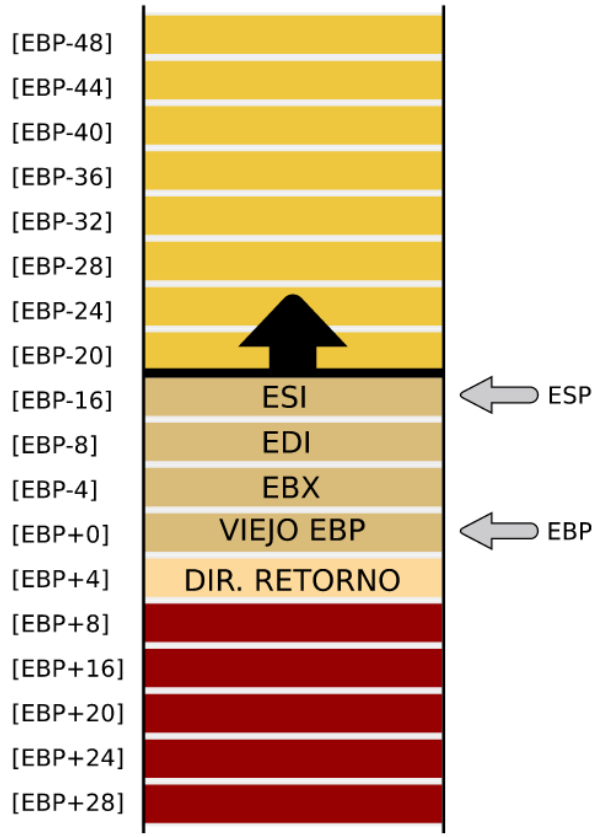


Caso 32 bits

- En este caso, las funciones que respetan la convención deben:
  - Preservar los registros `EBX`, `ESI` y `EDI`
  - Retornar el resultado a través de `EAX` (y `EDX` si ocupa 64b)
  - Preservar la consistencia de la pila
- Como antes, hay que atender a la información del resto de los registros que queramos preservar
- La pila está alineada a **4 bytes**, lo cual debemos preservar ante un llamado a función

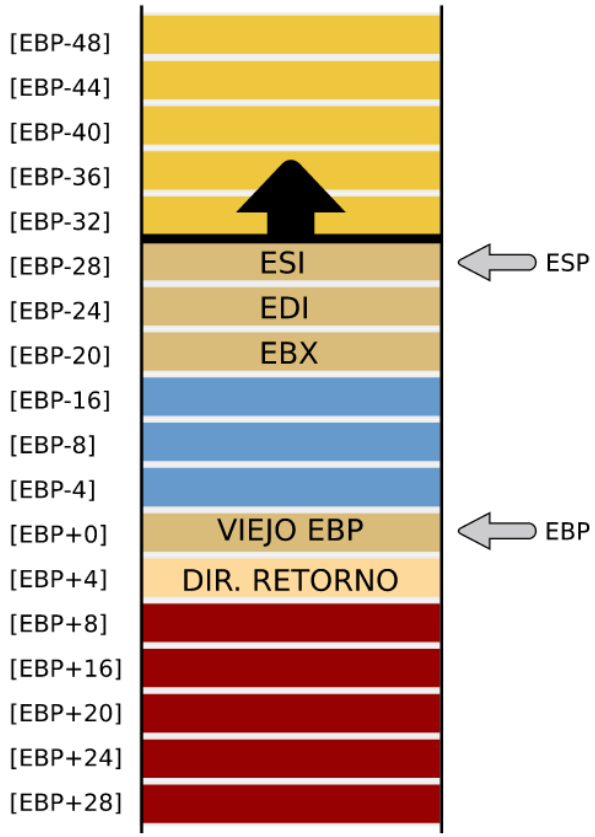
Estrcutura de un stack frame en 32 bits

```
fun:
  push ebp
  mov ebp, esp

  push ebx
  push edi
  push esi
  ... más código ...
  pop esi
  pop edi
  pop ebx

  pop ebp
  ret
```



Estrcutura de un stack frame en 32 bits

```
fun:
   push ebp
   mov ebp, esp
   sub esp, 12
   push ebx
   push edi
   push esi
   ... más código ...
   pop esi
   pop edi
   pop ebx
   add esp, 12
   pop ebp
   ret
```



Por convención System V AMD64 ABI

En 64 bits

Los parámetros se pasan de izquierda a derecha a través de:

- Los registros `RDI`, `RSI`, `RDX`, `RCX`, `R8` y `R9`
- Para números de punto **flotante**, los `XMM`s en orden ascendente

Si los registros no alcanzan, el resto de los argumentos se pasan a través de la pila pero en orden opuesto (de derecha a izquierda), para quedar ordenados desde la dirección más baja a la más alta

Por convención System V AMD64 ABI

En 64 bits

Los parámetros se pasan de izquierda a derecha a través de:

- Los registros `RDI`, `RSI`, `RDX`, `RCX`, `R8` y `R9`
- Para números de punto **flotante**, los `XMM`s en orden ascendente

Si los registros no alcanzan, el resto de los argumentos se pasan a través de la pila pero en orden opuesto (de derecha a izquierda), para quedar ordenados desde la dirección más baja a la más alta

En 32 bits

Los parámetros se pasan a través de la pila desde la dirección más baja a la más alta (se apilan de derecha a izquierda). Para valores de 64b se apila primero su parte alta.

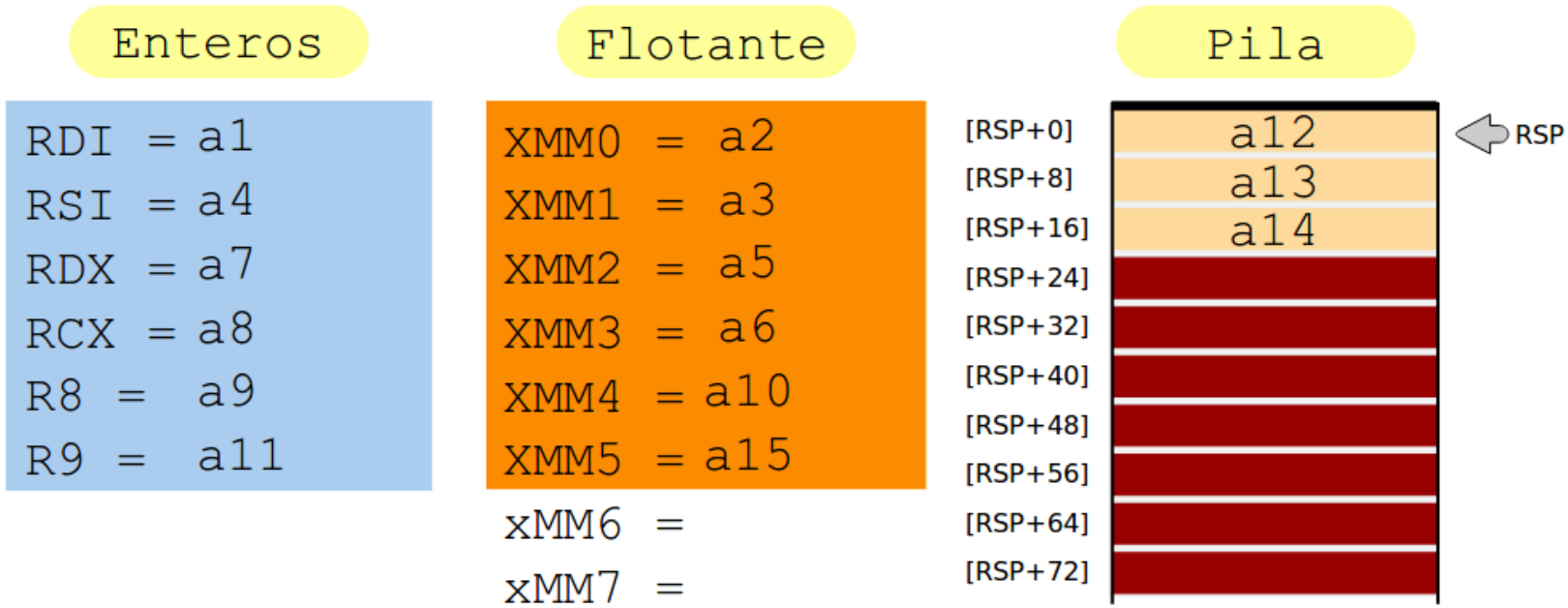
Ejemplo en 64 bits

```
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 =	

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)
```



Ejemplo en 32 bits

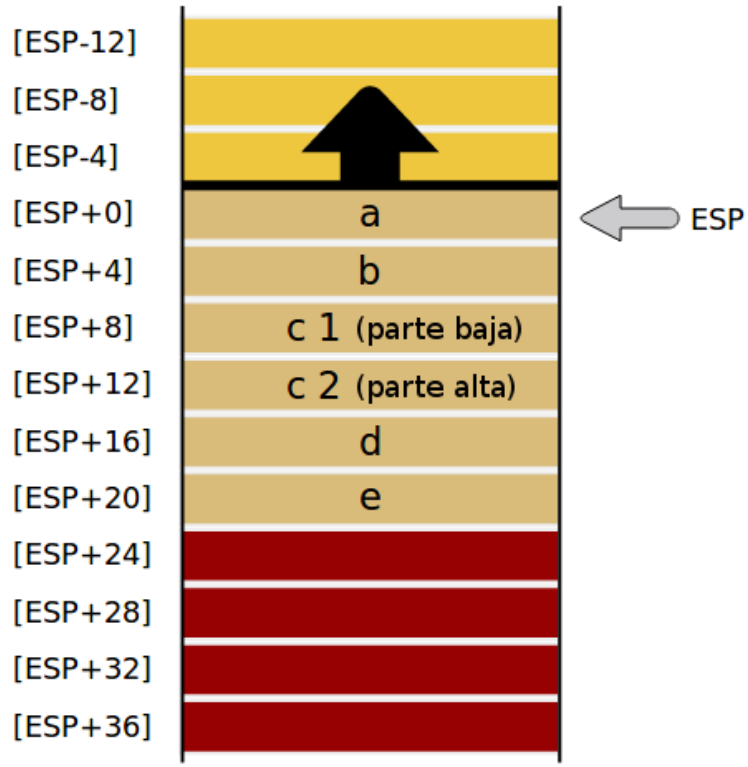
```
int f1( int a, float b, double c, int* d, double* e)
```

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 a funciones ASM desde C

Hacemos uso de la cláusula extern en C y global en ASM:

funcion.asm

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

programa.c

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

Llamar a funciones ASM desde C

Hacemos uso de la cláusula extern en C y global en ASM:

funcion.asm

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

programa.c

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

Primero ensamblamos y compilamos el código en ASM para luego linkearlo con el código en C:

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

Llamar funciones C desde ASM

Usamos sólo la cláusula extern en 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;

}
```

Compilamos ambos programas y generamos el ejecutable de ASM:

- `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

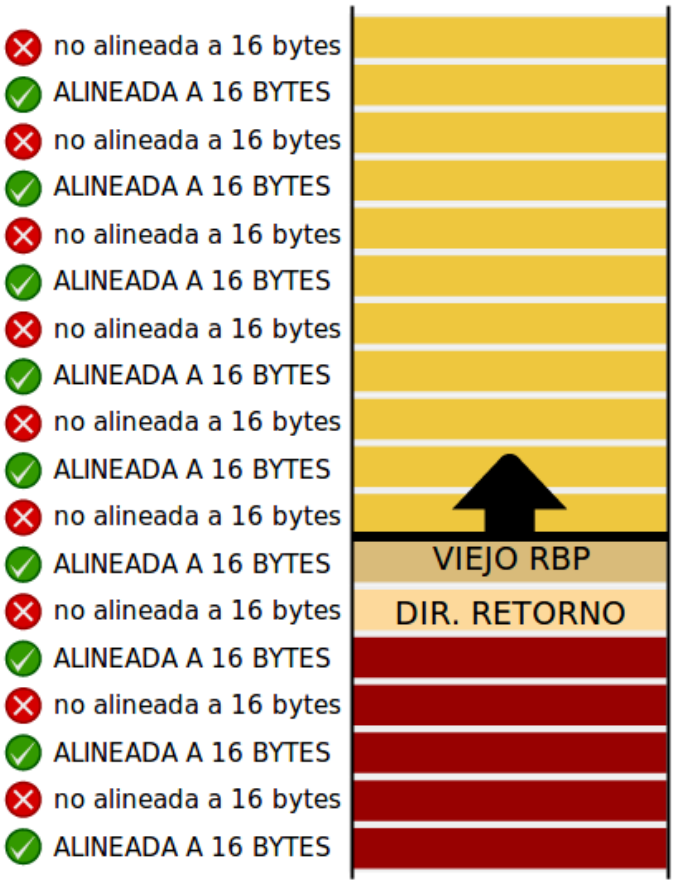
2. Modificar la función anterior para que sume dos numeros de tipo `double` (ver instrucción `ADDPD`)

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 );
```
Ésta retorna el resultado de la operación:
`a0-a1+a2-a3+a4-a5+a6-a7`

Alineación

Recuerden que al hacer cualquier llamada a función deben tener la pila alineada según la convención:



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

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

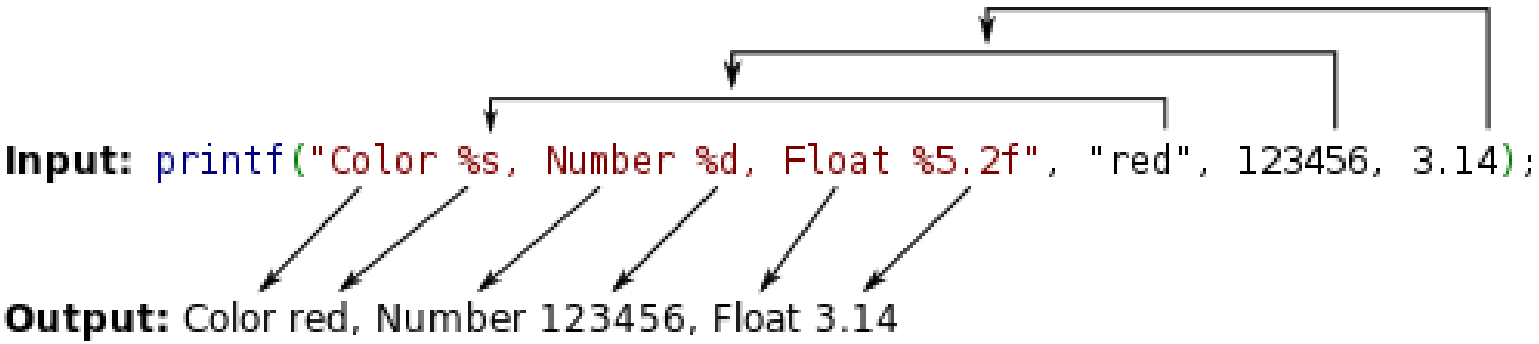
(3) - Cuando armamos el StackFrame guardamos el viejo RBP y alineamos la pila a 16 bytes

(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.

Funciones variádicas (de aridad variable)

- Para estas funciones debemos pasar la cantidad de números de puntos flotante por el registro AL



- En este caso, `RAX` vale 1

Fuentes y material adicional

- Artículo sobre las convenciones de llamadas a función en x86: https://en.wikipedia.org/wiki/X86_calling_conventions
- Artículo sobre System V ABI: https://wiki.osdev.org/System_V_ABI
- Documentación de NASM: https://nasm.us/doc/nasmdoc1.html
- Listado de opciones de comandos de ASM: https://docs.oracle.com/cd/E26502_01/html/E28388/assembler-19592.html

Eso es todo por hoy

¿Preguntas?