Pila y Convención C

David Alejandro González Márquez

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Temario

- Estructura y uso de la Pila

Temario

- Estructura y uso de la Pila
- Convención C
  - Stack Frame y conservación de registros
  - Pasaje de parámetros

Temario

- Estructura y uso de la Pila
- Convención C
  - Stack Frame y conservación de registros
  - Pasaje de parámetros
- Ensamblar, compilar y linkear, código C y ASM

Introducción

- La pila es una estructura en memoria.

Introducción

- La pila es una estructura en memoria.
- Se utiliza para guardar **información local** a una función.

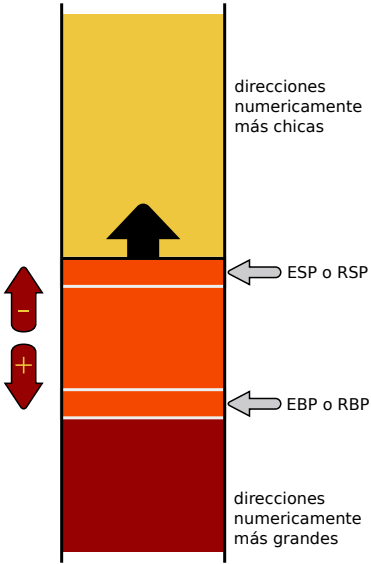
Introducción

- La pila es una estructura en memoria.
- Se utiliza para guardar **información local** a una función.
- Además sirve para almacenar información de **contexto**.

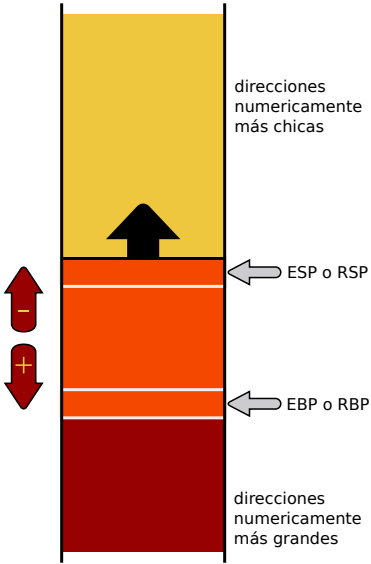
Introducción

- La pila es una estructura en memoria.
- Se utiliza para guardar **información local** a una función.
- Además sirve para almacenar información de **contexto**.
- Tanto en x86 como en x86-64, la pila tiene distintos parámetros.

Pila - Estructura



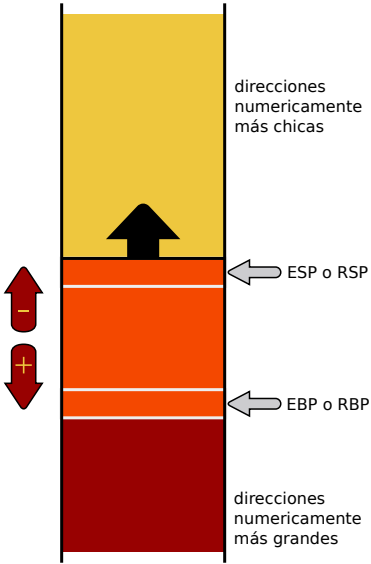
Pila - Estructura



En 32 bits

- Los registros EBP y ESP
- EBP (Base Pointer) apunta a la base
- ESP (Stack Pointer) al tope (último elemento válido)

Pila - Estructura



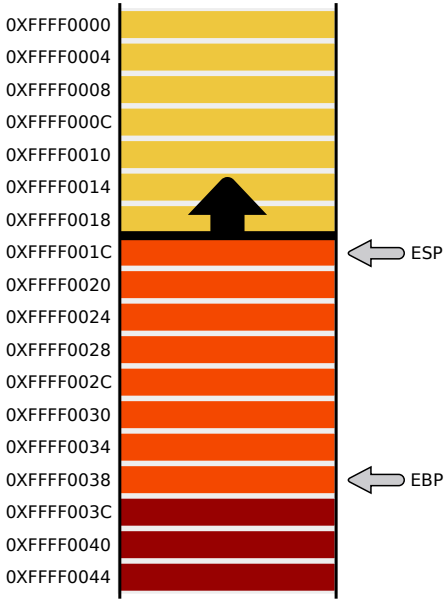
En 32 bits

- Los registros EBP y ESP
- EBP (Base Pointer) apunta a la base
- ESP (Stack Pointer) al tope (último elemento válido)

En 64 bits

- Los registros RBP y RSP
- RBP (Base Pointer) apunta a la base
- RSP (Stack Pointer) al tope (último elemento válido)

Pila en 32 bits - Estructura



Alineación

4 bytes (en 32bits)

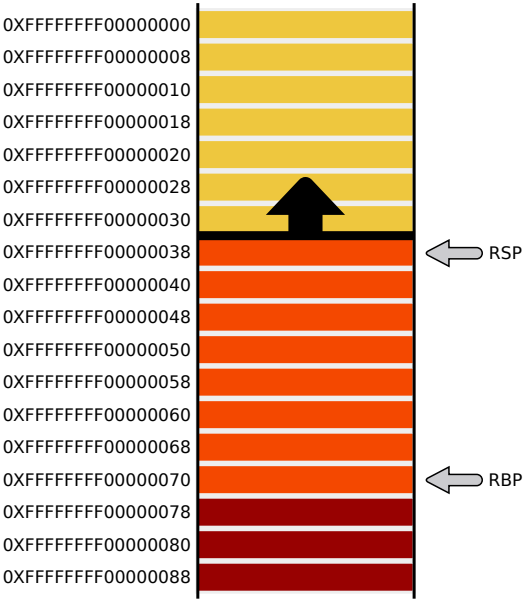
Instrucciones

- PUSH
- POP

Registros

- ESP
- EBP

Pila en 64 bits - Estructura



Alineación

8 bytes (en 64bits)

16 bytes para llamar funciones

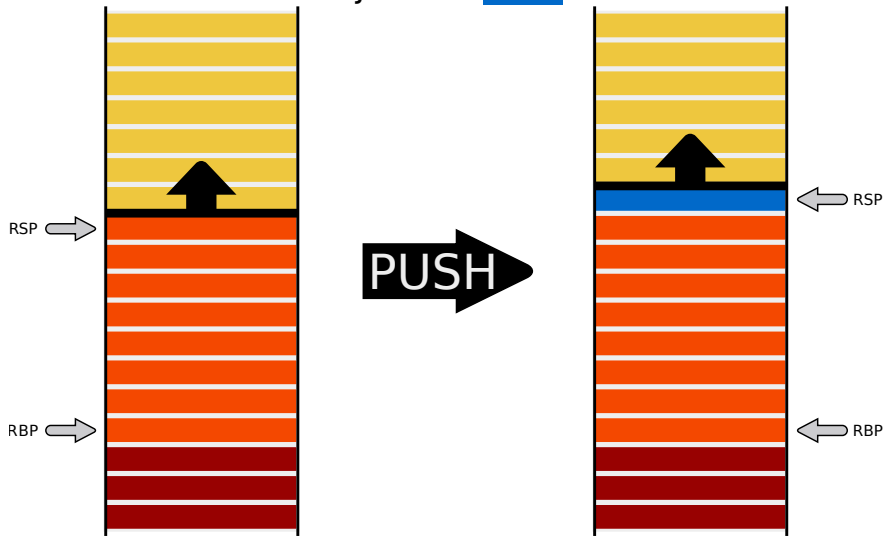
Instrucciones

- PUSH
- POP

Registros

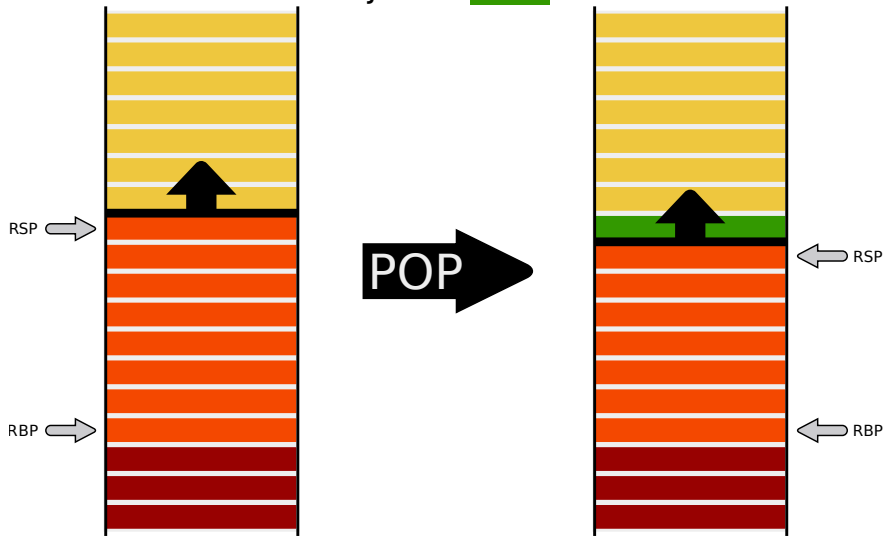
- RSP
- RBP

ej. PUSH RAX



Operaciones PUSH (apilar) y POP (desapilar)

ej. POP RAX



Operaciones PUSH (apilar) y POP (desapilar)

Convención C

- La forma en que se **codifican** los llamados a subrutinas en C es estática y depende de la **firma** de la función a llamar.

Convención C

- La forma en que se **codifican** los llamados a subrutinas en C es estática y depende de la **firma** de la función a llamar.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.

Convención C

- La forma en que se **codifican** los llamados a subrutinas en C es estática y depende de la **firma** de la función a llamar.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.

- La convención define:
  - Cómo las funciones **reciben parámetros**.

Convención C

- La forma en que se **codifican** los llamados a subrutinas en C es estática y depende de la **firma** de la función a llamar.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.

- La convención define:
  - Cómo las funciones **reciben parámetros**.
  - Cómo las funciones **retornan el resultado**.

Convención C

- La forma en que se **codifican** los llamados a subrutinas en C es estática y depende de la **firma** de la función a llamar.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.

- La convención define:
  - Cómo las funciones **reciben parámetros**.
  - Cómo las funciones **retornan el resultado**.
  - Qué registros se **deben preservar** en una función.

Convención C

- La forma en que se **codifican** los llamados a subrutinas en C es estática y depende de la **firma** de la función a llamar.

- De esta forma las funciones pueden ser llamadas sin tener en cuenta como fueron implementadas.

- La convención define:
  - Cómo las funciones **reciben parámetros**.
  - Cómo las funciones **retornan el resultado**.
  - Qué registros se **deben preservar** en una función.

- Las convenciones dependen de la arquitectura del procesador y del sistema operativo:
  - En `x86/Linux` (32bits) se conoce como `x32 ABI`.
  - En `x86-64/Linux` (64bits) se deomina `System V AMD64 ABI`.

Stack frame

Una función en C es ejecutada dentro de un **contexto de ejecución**, esté contiene un puntero válido al tope de pila y un puntero a base de pila.

Stack frame

Una función en C es ejecutada dentro de un **contexto de ejecución**, esté contiene un puntero válido al tope de pila y un puntero a base de pila.

stack frame

Estructura en memoria constituida por la dirección de retorno, el conjunto de registros preservados, las variables locales y los parámetros pasados por pila.

Stack frame

Una función en C es ejecutada dentro de un **contexto de ejecución**, esté contiene un puntero válido al tope de pila y un puntero a base de pila.

stack frame

Estructura en memoria constituida por la dirección de retorno, el conjunto de registros preservados, las variables locales y los parámetros pasados por pila.

La construcción del *stack frame* consiste en colocar el registro base de la pila en una dirección relativa al comienzo del área de la función llamadora.

Caso 32 bits

- Preservar los registros EBX, ESI, EDI y EBP
  (Se deben preservar SOLO los registros que se modifican).

Caso 32 bits

- Preservar los registros EBX, ESI, EDI y EBP
  (Se deben preservar SOLO los registros que se modifican).
- Retornar el resultado a través de EAX (y EDX : EAX si ocupa 64 bits).

Caso 32 bits

- Preservar los registros EBX, ESI, EDI y EBP
  (Se deben preservar SOLO los registros que se modifican).
- Retornar el resultado a través de EAX (y EDX : EAX si ocupa 64 bits).
- Preservar la consistencia de la pila.

Caso 32 bits

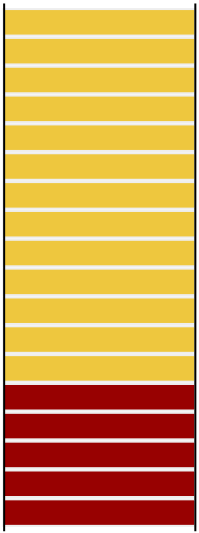
- Preservar los registros EBX, ESI, EDI y EBP
  (Se deben preservar SOLO los registros que se modifican).
- Retornar el resultado a través de EAX (y EDX : EAX si ocupa 64 bits).
- Preservar la consistencia de la pila.
- Los parámetros se pasan por pila.

Caso 32 bits

- Preservar los registros EBX, ESI, EDI y EBP
  (Se deben preservar SOLO los registros que se modifican).
- Retornar el resultado a través de EAX (y EDX : EAX si ocupa 64 bits).
- Preservar la consistencia de la pila.
- Los parámetros se pasan por pila.
- La pila debe estar alineada a 4 bytes antes de un llamado a función.

Caso 32 bits

fun:



- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

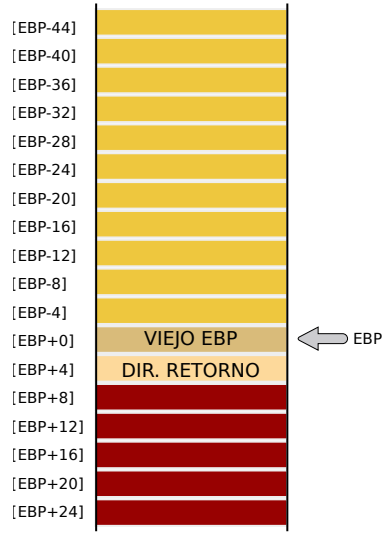
Caso 32 bits

fun:

- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

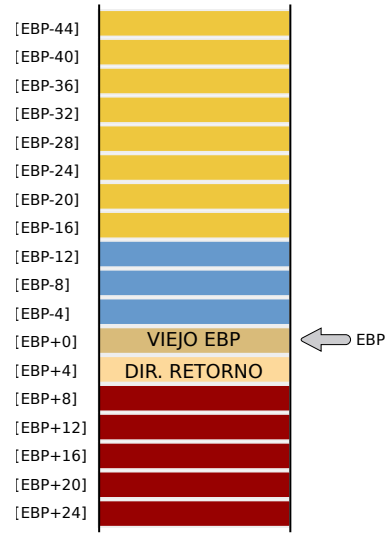
```
fun:
        PUSH EBP
        MOV  EBP,ESP
```



- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

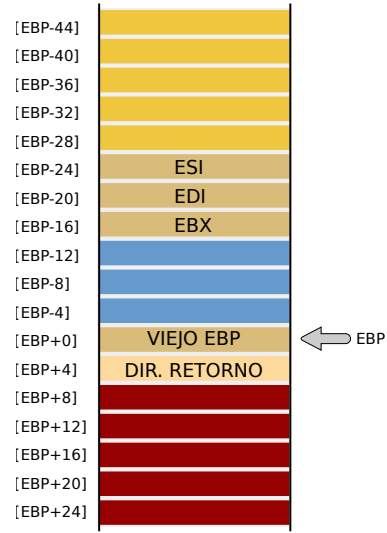
```
fun:
        PUSH EBP
        MOV  EBP,ESP
        SUB ESP,12
```



- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

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

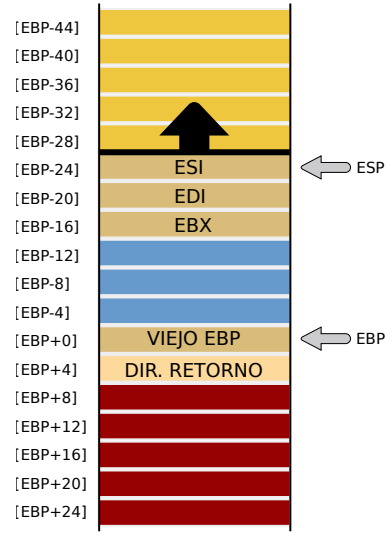


- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

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

MI CÓDIGO



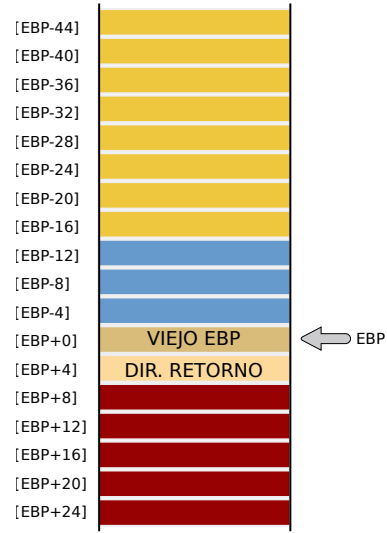
- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

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

    MI
    CÓDIGO

    POP ESI
    POP EDI
    POP EBX
```



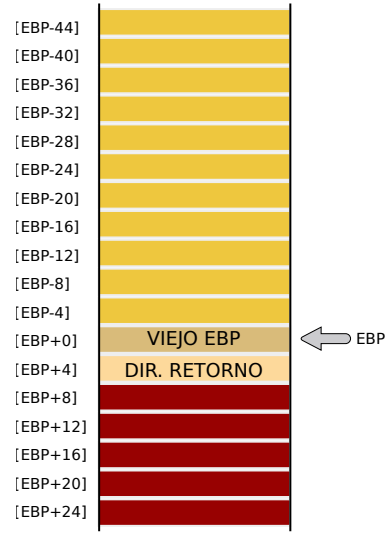
- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

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

        MI
        CÓDIGO

        POP ESI
        POP EDI
        POP EBX
        ADD ESP,12
```



- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

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

        MI
        CÓDIGO

        POP ESI
        POP EDI
        POP EBX
        ADD ESP,12
        POP EBP
```

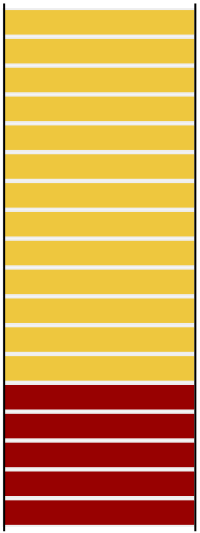
- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

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

        MI
        CÓDIGO

        POP ESI
        POP EDI
        POP EBX
        ADD ESP,12
        POP EBP
        RET
```



- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 64 bits

- Preservar los registros RBX, R12, R13, R14, R15 y RBP
  (Se deben preservar SOLO los registros que se modifican).

Caso 64 bits

- Preservar los registros RBX, R12, R13, R14, R15 y RBP
  (Se deben preservar SOLO los registros que se modifican).
- Retornar el resultado a través de RAX si es un valor entero
  (y RDX : RAX si ocupa 128bits) o XMM0, si es un número de punto flotante.

Caso 64 bits

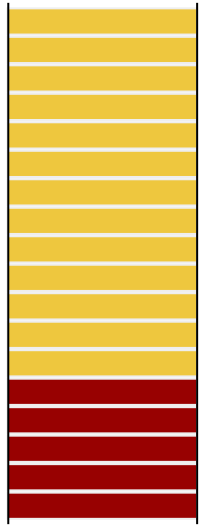
- Preservar los registros RBX, R12, R13, R14, R15 y RBP
  (Se deben preservar SOLO los registros que se modifican).
- Retornar el resultado a través de RAX si es un valor entero
  (y RDX : RAX si ocupa 128bits) o XMM0, si es un número de punto flotante.
- Preservar la consistencia de la pila.

Caso 64 bits

- Preservar los registros RBX, R12, R13, R14, R15 y RBP
  (Se deben preservar SOLO los registros que se modifican).
- Retornar el resultado a través de RAX si es un valor entero
  (y RDX : RAX si ocupa 128bits) o `XMM0`, si es un número de punto flotante.
- Preservar la consistencia de la pila.
- La pila opera alineada a 8 bytes.
  Pero antes de llamar a funciones de C debe estarlo a **16 bytes**.

Caso 64 bits

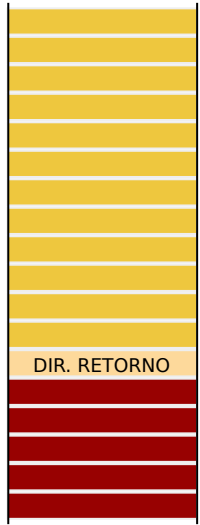
fun:



- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

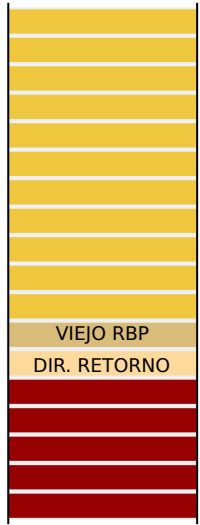
fun:



- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

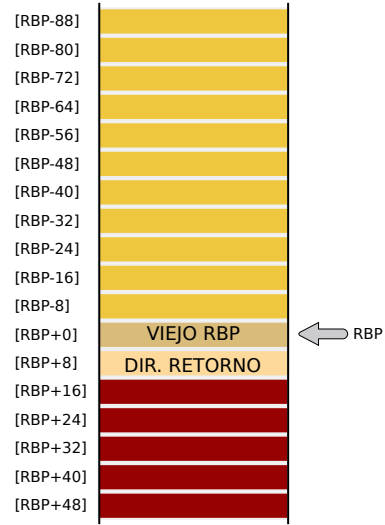
```
fun:
        PUSH RBP
```



- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

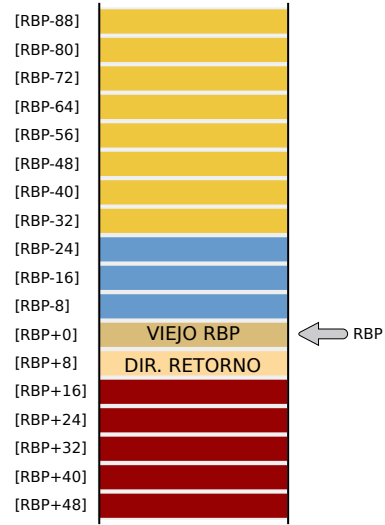
```
fun:
        PUSH RBP
        MOV  RBP,RSP
```



- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

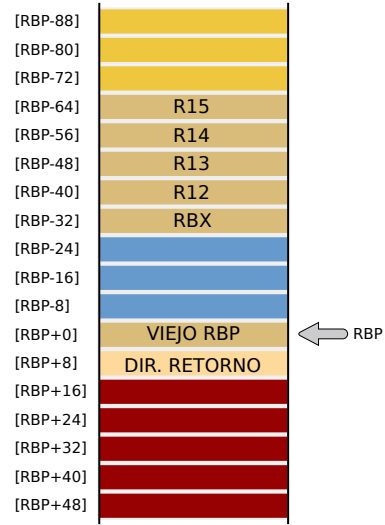
```
fun:
        PUSH RBP
        MOV  RBP,RSP
        SUB RSP,24
```



- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

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

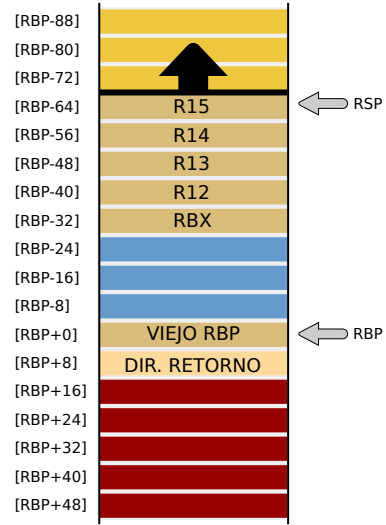


- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

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

MI CÓDIGO



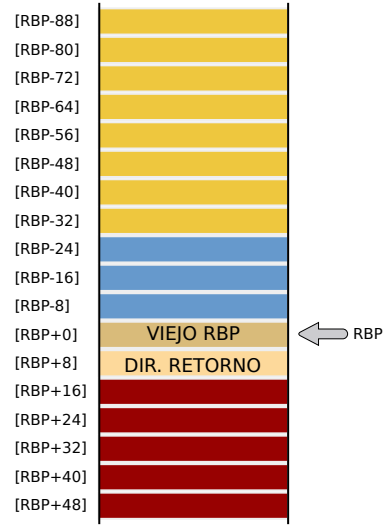
- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

```
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
```



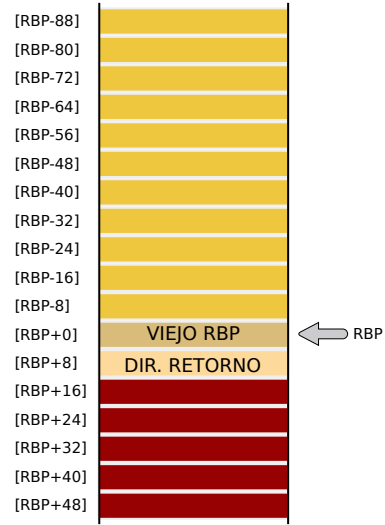
- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

```
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
```



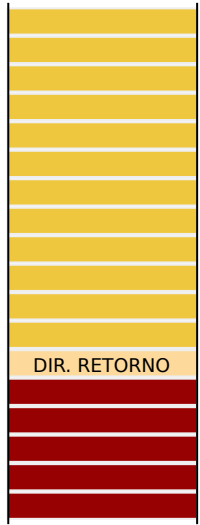
- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

```
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
```



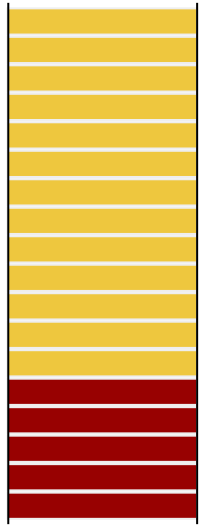
- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

```
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
```



- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Pasaje de parámetros

En 32 bits

- Los parámetros se pasan **a través de la pila** desde la dirección más baja a la más alta.

Pasaje de parámetros

En 32 bits

- Los parámetros se pasan **a través de la pila** desde la dirección más baja a la más alta.
- Son apilados de derecha a izquierda según aparecen en la firma de la función.

Pasaje de parámetros

En 32 bits

- Los parámetros se pasan **a través de la pila** desde la dirección más baja a la más alta.
- Son apilados de derecha a izquierda según aparecen en la firma de la función.
- Para valores de 64bits se apilan en little-endian.

Pasaje de parámetros

En 32 bits

- Los parámetros se pasan **a través de la pila** desde la dirección más baja a la más alta.
- Son apilados de derecha a izquierda según aparecen en la firma de la función.
- Para valores de 64bits se apilan en little-endian.

En 64 bits

- Los parámetros **se pasan por registro**, de izquierda a derecha según la firma de la función, clasificados por tipo:

Pasaje de parámetros

En 32 bits

- Los parámetros se pasan **a través de la pila** desde la dirección más baja a la más alta.
- Son apilados de derecha a izquierda según aparecen en la firma de la función.
- Para valores de 64bits se apilan en little-endian.

En 64 bits

- Los parámetros **se pasan por registro**, de izquierda a derecha según la firma de la función, clasificados por tipo:
  - Enteros y direcciónes de memoria: `RDI`, `RSI`, `RDX`, `RCX`, `R8` y `R9`

Pasaje de parámetros

En 32 bits

- Los parámetros se pasan **a través de la pila** desde la dirección más baja a la más alta.
- Son apilados de derecha a izquierda según aparecen en la firma de la función.
- Para valores de 64bits se apilan en little-endian.

En 64 bits

- Los parámetros **se pasan por registro**, de izquierda a derecha según la firma de la función, clasificados por tipo:
  - Enteros y direcciónes de memoria: `RDI`, `RSI`, `RDX`, `RCX`, `R8` y `R9`
  - Punto flotante: `XMM0` a `XMM7`

Pasaje de parámetros

En 32 bits

- Los parámetros se pasan **a través de la pila** desde la dirección más baja a la más alta.
- Son apilados de derecha a izquierda según aparecen en la firma de la función.
- Para valores de 64bits se apilan en little-endian.

En 64 bits

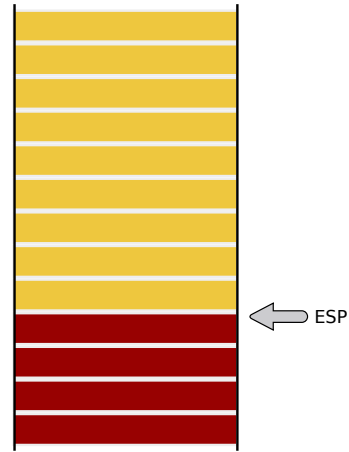
- Los parámetros **se pasan por registro**, de izquierda a derecha según la firma de la función, clasificados por tipo:
  - Enteros y direcciónes de memoria: `RDI`, `RSI`, `RDX`, `RCX`, `R8` y `R9`
  - Punto flotante: `XMM0` a `XMM7`
  - Resto de los parámetros que superen la cantidad de registros se ubican en la pila como en 32 bits.

Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c0
push b
push a
call f1
add esp, 6*4
...
```

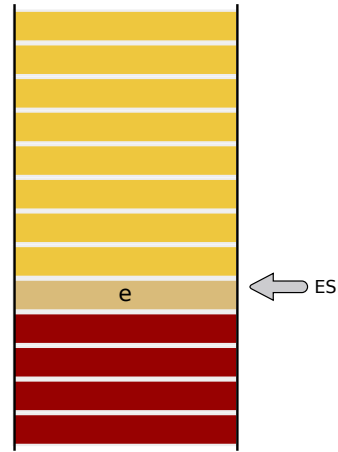


Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c0
push b
push a
call f1
add esp, 6*4
...
```

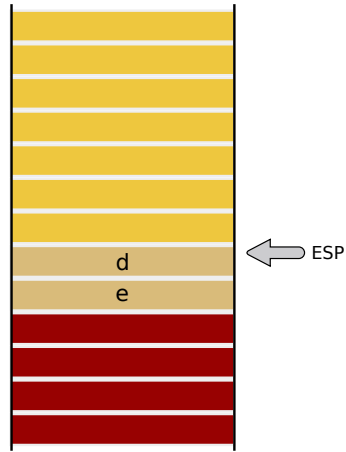


Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c0
push b
push a
call f1
add esp, 6*4
...
```

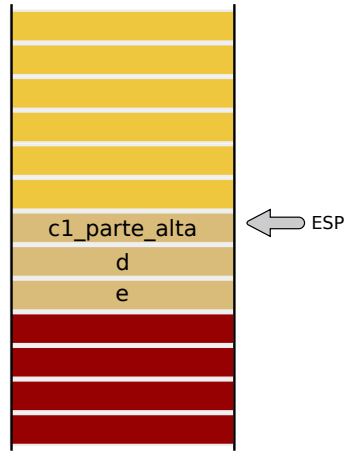


Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c0
push b
push a
call f1
add esp, 6*4
...
```

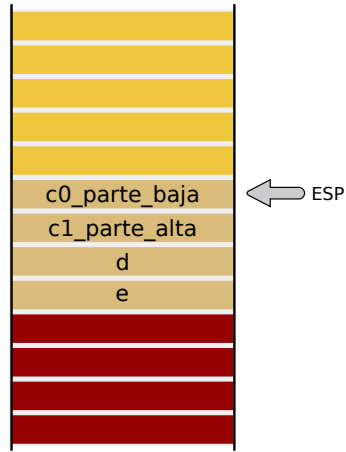


Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c0
push b
push a
call f1
add esp, 6*4
...
```

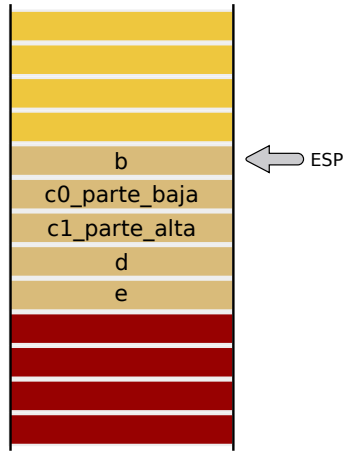


Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c0
push b
push a
call f1
add esp, 6*4
...
```

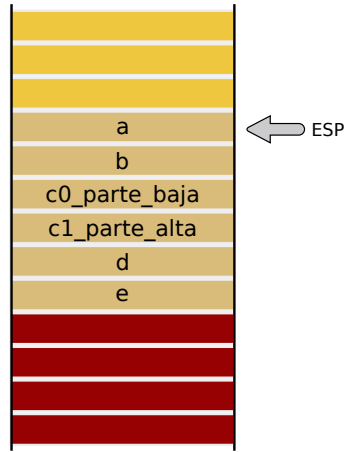


Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c0
push b
push a
call f1
add esp, 6*4
...
```

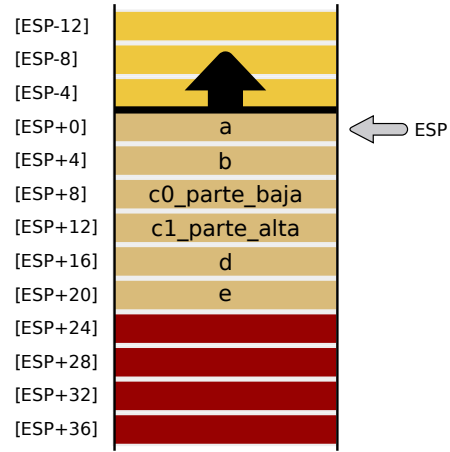


Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c0
push b
push a
call f1
add esp, 6*4
...
```

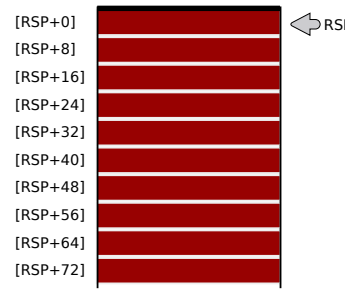


Ejemplo en 64 bits

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

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

Pila

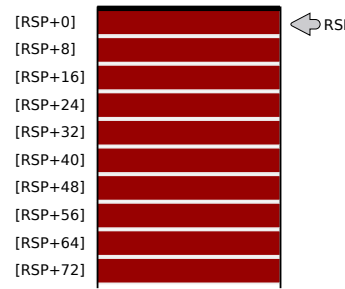


Ejemplo en 64 bits

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

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

Pila

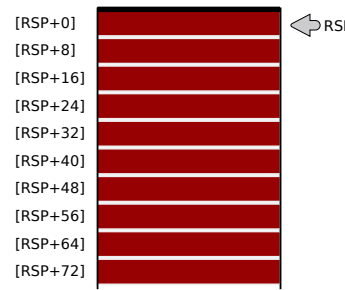


Ejemplo en 64 bits

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

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

Pila

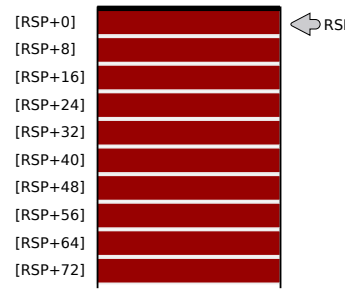


Ejemplo en 64 bits

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

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

Pila



Ejemplo en 64 bits

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

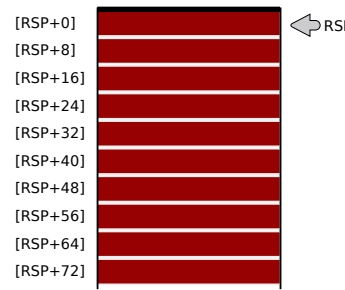
Enteros

```
RDI = a
RSI = d
RDX =
RCX =
R8  =
R9  =
```

Flotante

```
XMM0 = b
XMM1 = c
XMM2 =
XMM3 =
XMM4 =
XMM5 =
xMM6 =
xMM7 =
```

Pila



Ejemplo en 64 bits

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

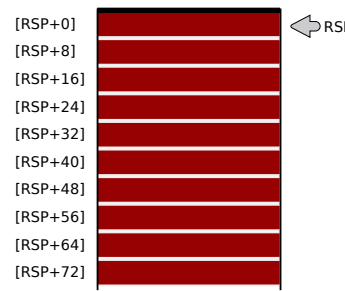
Enteros

```
RDI = a
RSI = d
RDX = e
RCX =
R8  =
R9  =
```

Flotante

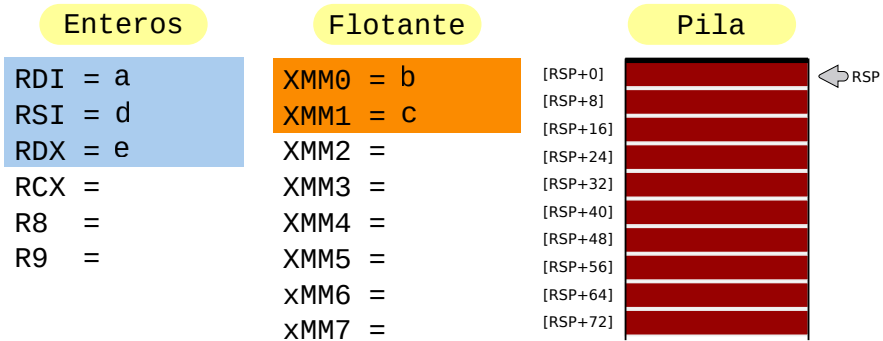
```
XMM0 = b
XMM1 = c
XMM2 =
XMM3 =
XMM4 =
XMM5 =
xMM6 =
xMM7 =
```

Pila



Ejemplo en 64 bits

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



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)
```

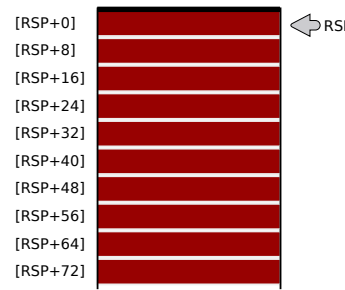
Enteros

RDI =
RSI =
RDX =
RCX =
R8 =
R9 =

Flotante

XMM0 =
XMM1 =
XMM2 =
XMM3 =
XMM4 =
XMM5 =
xMM6 =
xMM7 =

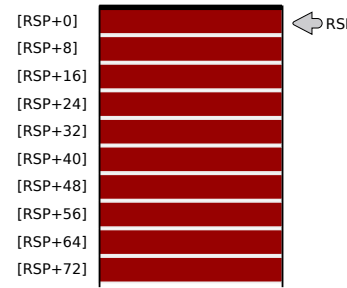
Pila



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 = a1	XMM0 =	[RSP+0] ← RSP
RSI =	XMM1 =	[RSP+8]
RDX =	XMM2 =	[RSP+16]
RCX =	XMM3 =	[RSP+24]
R8 =	XMM4 =	[RSP+32]
R9 =	XMM5 =	[RSP+40]
	xMM6 =	[RSP+48]
	xMM7 =	[RSP+56]
		[RSP+64]
		[RSP+72]



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

RDI = a1
RSI =
RDX =
RCX =
R8 =
R9 =

Flotante

XMM0 = a2
XMM1 =
XMM2 =
XMM3 =
XMM4 =
XMM5 =
xMM6 =
xMM7 =

Pila

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)
```

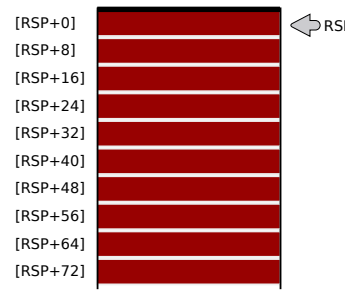
Enteros

```
RDI = a1
RSI =
RDX =
RCX =
R8  =
R9  =
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 =
XMM3 =
XMM4 =
XMM5 =
xMM6 =
xMM7 =
```

Pila



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)
```

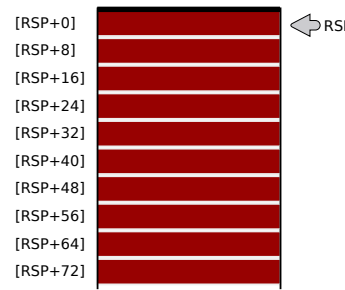
Enteros

RDI = a1
RSI = a4
RDX =
RCX =
R8 =
R9 =

Flotante

XMM0 = a2
XMM1 = a3
XMM2 =
XMM3 =
XMM4 =
XMM5 =
xMM6 =
xMM7 =

Pila



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)
```

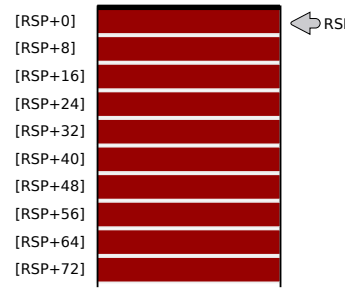
Enteros

```
RDI = a1
RSI = a4
RDX =
RCX =
R8  =
R9  =
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 =
XMM4 =
XMM5 =
xMM6 =
xMM7 =
```

Pila



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)
```

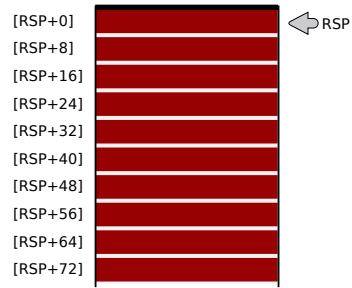
Enteros

RDI = a1
RSI = a4
RDX =
RCX =
R8 =
R9 =

Flotante

XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 =
XMM5 =
xMM6 =
xMM7 =

Pila



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)
```

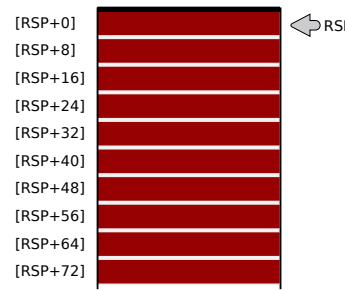
Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX =
R8  =
R9  =
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 =
XMM5 =
xMM6 =
xMM7 =
```

Pila



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)
```

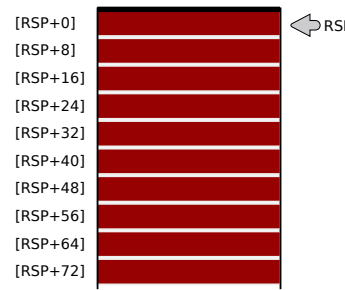
Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  =
R9  =
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 =
XMM5 =
xMM6 =
xMM7 =
```

Pila



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)
```

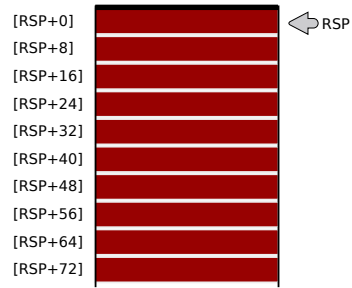
Enteros

RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8 = a9
R9 =

Flotante

XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 =
XMM5 =
xMM6 =
xMM7 =

Pila



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)
```

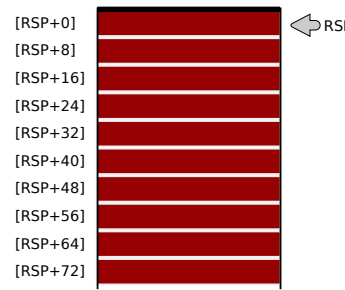
Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  = a9
R9  =
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 = a10
XMM5 =
xMM6 =
xMM7 =
```

Pila



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)
```

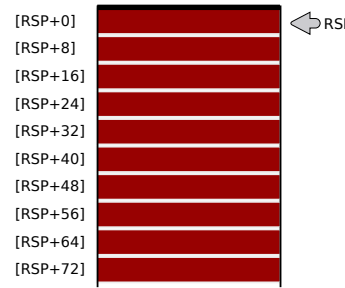
Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  = a9
R9  = a11
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 = a10
XMM5 =
xMM6 =
xMM7 =
```

Pila



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)
```

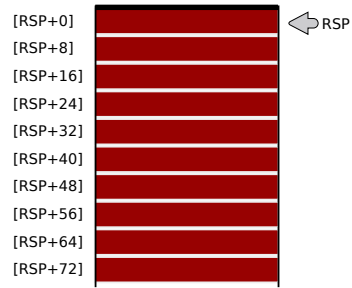
Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  = a9
R9  = a11
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 = a10
XMM5 = a15
xMM6 =
xMM7 =
```

Pila



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)
```

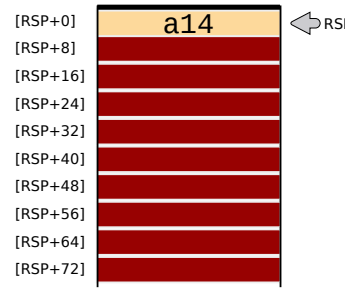
Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  = a9
R9  = a11
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 = a10
XMM5 = a15
xMM6 =
xMM7 =
```

Pila



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)
```

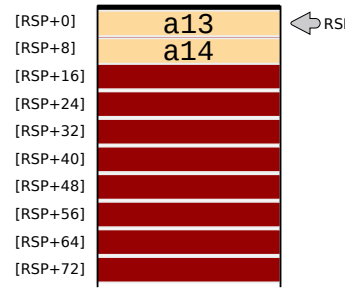
Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  = a9
R9  = a11
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 = a10
XMM5 = a15
xMM6 =
xMM7 =
```

Pila



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)
```

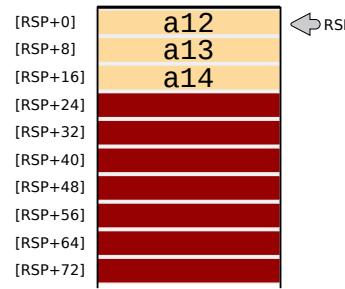
Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  = a9
R9  = a11
```

Flotante

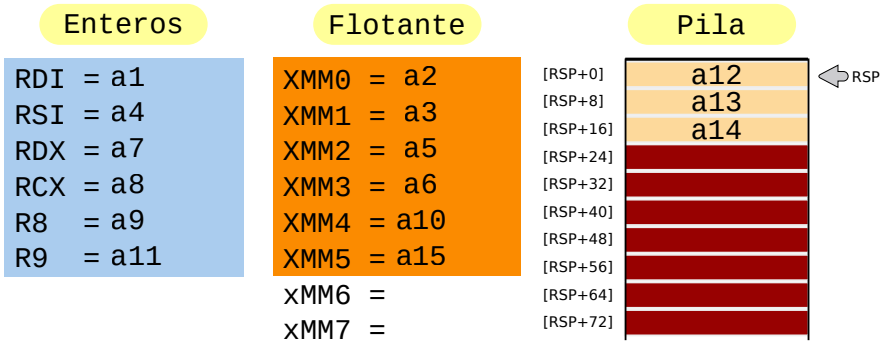
```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 = a10
XMM5 = a15
xMM6 =
xMM7 =
```

Pila



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)
```



Interacción C-ASM - **Llamar a funciones ASM desde C**

`global` indica que el simbolo `fun` es visible desde el exterior del ASM.
`extern` permite declarar la firma `fun` para luego ser linkeada.

funcion.asm

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

programa.c

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

Interacción C-ASM - **Llamar a funciones ASM desde C**

`global` indica que el simbolo `fun` es visible desde el exterior del ASM.
`extern` permite declarar la firma `fun` para luego ser linkeada.

funcion.asm

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

programa.c

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

1. Ensamblar código ASM:
   `nasm -f elf64 funcion.asm -o funcion.o`
2. Compilar y linkear el código C:
   `gcc -o ejec programa.c funcion.o`

Interacción C-ASM - **Llamar funciones C desde ASM**

`extern` indica que el simbolo no esta definido en el 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;

}
```

Interacción C-ASM - **Llamar funciones C desde ASM**

`extern` indica que el simbolo no esta definido en el 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;

}
```

1. Ensamblar código ASM:
   `nasm -f elf64 main.asm -o main.o`
2. Compilar código C en un objeto
   `gcc -no-pie -c -m64 funcion.c -o funcion.o`
3. Usar gcc como linker de ambos archivos objeto.
   `gcc -no-pie -o ejec -m64 main.o funcion.o`

Notas: Alineación

1. Antes de llamar a una función la pila debe estar alineada a 16 bytes.

Notas: Alineación

1. Antes de llamar a una función la pila debe estar alineada a 16 bytes.

2. Al entrar a una función, se guarda en la pila la dirección de retorno, por lo tanto queda desalineada. Ya que la dirección de retorno ocupa un lugar en la pila, es decir 8 bytes.

Notas: Alineación

1. Antes de llamar a una función la pila debe estar alineada a 16 bytes.
2. Al entrar a una función, se guarda en la pila la dirección de retorno, por lo tanto queda desalineada. Ya que la dirección de retorno ocupa un lugar en la pila, es decir 8 bytes.
3. Al ejecutar `push rbp` la pila vuelve a quedar alineada a 16 bytes.

Notas: Alineación

1. Antes de llamar a una función la pila debe estar alineada a 16 bytes.
2. Al entrar a una función, se guarda en la pila la dirección de retorno, por lo tanto queda desalineada. Ya que la dirección de retorno ocupa un lugar en la pila, es decir 8 bytes.
3. Al ejecutar `push rbp` la pila vuelve a quedar alineada a 16 bytes.

Recordar alinear la pila a 16 bytes antes de llamar a una función.
Esto se debe realizar por convención.

Notas: Funciones variádicas (de aridad variable)

Notas: Funciones variádicas (de aridad variable)

Para estas funciones toman una **cantidad variable de parámetros**. Depende como estén implementadas, identifican la cantidad de parámetros pasados.

Notas: Funciones variádicas (de aridad variable)

Para estas funciones toman una **cantidad variable de parámetros**. Depende como estén implementadas, identifican la cantidad de parámetros pasados.

Caso: `printf`

```
printf(char* formato, ...)
```

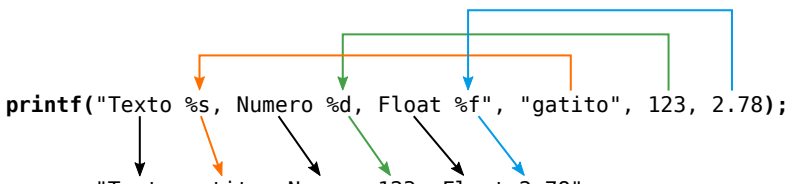
Notas: Funciones variádicas (de aridad variable)

Para estas funciones toman una **cantidad variable de parámetros**. Depende como estén implementadas, identifican la cantidad de parámetros pasados.

Caso: `printf`

```
printf(char* formato, ...)
```

Ejemplo:

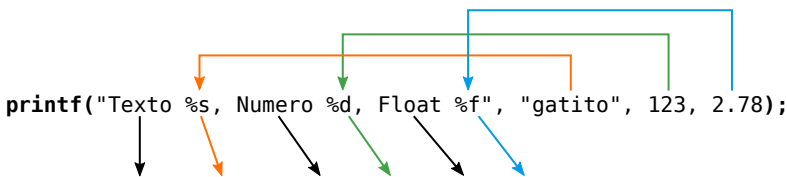


Notas: Funciones variádicas (de aridad variable)

Para estas funciones toman una **cantidad variable de parámetros**. Depende como estén implementadas, identifican la cantidad de parámetros pasados.

Caso: `printf`

```
printf(char* formato, ...)
```



Ejemplo:

IMPORTANTE: Desde ASM

Se debe pasar en RAX el número 1, si se va a imprimir valores en punto flotante.

Además: Usar el flag `-no-pie` para que el linker use símbolos estáticos (ver bibliográfica)

Ejercicios

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

Bibliografía: Fuentes y material adicional

- Convenciones de llamados a función en x86:
  `https://en.wikipedia.org/wiki/X86_calling_conventions`
- Notas sobre System V ABI:
  `https://wiki.osdev.org/System_V_ABI`
- Documentación de NASM:
  `https://nasm.us/doc/`
- Artículo sobre el flag `-pie`:
  `https://eklitzke.org/position-independent-executables`
- Documentación de System V ABI:
  `https://uclibc.org/docs/psABI-x86_64.pdf`
- Manuales de Intel:
  `https://software.intel.com/en-us/articles/intel-sdm`

¡Gracias!

Recuerden leer los comentarios al final de este video por aclaraciones o fe de erratas.