Organización del Computador II

David Alejandro González Márquez

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

13-08-2019

Bienvenidos a ORGA II

Profesor

- Alejandro Furfaro

JTPs

- David González Márquez
- Iván Arcuschin

Ay2

- Facundo Ruiz
- Belen Ticona
- Facundo Linari
- Sofia Massobrio
- Ezequiel Barrios








Clases

Teóricas

Jueves de 17 a 19 hs
Aula 2

Prácticas

Martes de 17 a 22 hs
Aula 2 y Labos 6/7
Jueves de 19 a 22 hs
Labos 6/7

Evaluación

Trabajos Prácticos

TP1: Indiv. - 05/09 - 26/09
TP2: Grupal - 10/09 - 03/12
TP3: Grupal - 07/11 - 03/12

Parciales

1er Parcial: 01/10
2do Parcial: 19/11
1er Recuperatorio: 26/11
2do Recuperatorio: 03/12

Régimen de Aprobación

Parciales

Calificaciones: I (0 a 59), A- (60 a 64) y A (65 a 100)
No pueden aprobar con A- ambos parciales
Los recuperatorios tienen 2 notas: I (0 a 64) y A (65 a 100)

Trabajos Prácticos

Calificaciones: I, A
TP1 → individual (sin informe)
TP2 y TP3 → (con informe) grupos de 3 personas
Entregas mediante GIT

Aprobar Trabajos Prácticos

Aprobar parciales
Aprobar TPs

$\Rightarrow$

Materia

Aprobar Final

Regímenes especiales

Trabajo Práctico Final

Más de 70 en ambos parciales (no recuperatorios)
y habiendo aprobado los trabajos prácticos en primera instancia
Posibilidad de hacer un tp final.

Extensión de Aprobación de TPs

Tener aprobados los 3 TPs
Mediante solicitud y coloquio individual
Se salvan los Tps **por un sólo cuatrimestre**

Medios de Comunicación

- **Página de la materia**
  `https://campus.exactas.uba.ar/`

- **Lista de docentes**
  `orga2-doc@dc.uba.ar`
  Consultas, sugerencias, quejas, agradecimientos, insultos, etc

- **Lista de alumnos**
  `orga2-alu@dc.uba.ar`
  Uso casi exclusivo para envío de mensajes a los alumnos
  Vale el "busco grupo" o "el sábado por la noche sale tp"

- **No oficial**
  `#Orga2 @ freenode.net` (IRC)
  `https://t.me/joinchat/Cy7ktORKuAeYfsyffvTR2A` (Telegram)

Bibliografía

Arquitectura

- Manuales de Intel
  (se los pueden bajar de la página de la materia)
- The Unabridged Pentium 4: IA32 Processor Genealogy
  MindShare, Tom Shamley, INC. Addison-Wesley
- Computer Architecture: A Quantitative Approach, 4th Edition
  John L. Hennessy , David A. Patterson

Interacción con lenguajes de alto nivel

- Thinking in C, Volumen 1; Bruce Eckel; Mindview, Inc.
- Programming Languages: Design and Implementation, 4/E; Terrence W. Pratt, Marvin V. Zelkowitz

¡Bienvenidos!

Ejercicio

Describa en sus palabras las funciones de las siguientes aplicaciones:

- Compilador
- Ensamblador
- Linker

Ejercicio

Describa en sus palabras las funciones de las siguientes aplicaciones:

- Compilador
- Ensamblador
- Linker

- **Compilador**: Toma código en un lenguaje de alto nivel y lo transforma a código ensamblador de alguna arquitectura.
- **Ensamblador**: Toma código en lenguaje ensamblador y lo traduce a código de máquina, generando un archivo objeto. Resuelve nombres, simbólicos y traduce los mnemónicos.
- **Linker**: Toma varios archivos objeto y los transforma en un ejecutable.

Ejercicio

Muestre cómo se almacenan en memoria los siguientes datos en procesadores Big-Endian y Little-Endian:

```
DB 12h                DD 12345678h
DB 12h, 34h           DD 12345678h, 9ABCDEF1h
DW 1234h              DQ 123456789ABCDEF1h
DW 1234h, 5678h       DB '1234'
```

Ejercicio

Muestre cómo se almacenan en memoria los siguientes datos en procesadores Big-Endian y Little-Endian:

```
DB 12h                DD 12345678h
DB 12h, 34h           DD 12345678h, 9ABCDEF1h
DW 1234h              DQ 123456789ABCDEF1h
DW 1234h, 5678h       DB '1234'
```

`DB`, `DW`, `DD`, `DQ`: Pseudo-instrucciones para el ensamblador que indican cómo definir datos en el archivo objeto.
NO se ejecutan por la CPU, las interpreta el ensamblador.

Big Endian: el byte más significativo en la posición de memoria menos significativa.
Little Endian: el byte más significativo en la posición de memoria más significativa.

`DB 12h`	`-|12|-` big endian `-|12|-` little endian
`DB 12h, 34h`	`-|12|34|-` big endian `-|12|34|-` little endian
`DW 1234h`	`-|12|34|-` big endian `-|34|12|-` little endian
`DW 1234h, 5678h`	`-|12|34|56|78|-` big endian `-|34|12|78|56|-` little endian
`DD 12345678h`	`-|12|34|56|78|-` big endian `-|78|56|34|12|-` little endian
`DD 12345678h, 9ABCDEF1h`	`-|12|34|56|78|9A|BC|DE|F1|-` big endian `-|78|56|34|12|F1|DE|BC|9A|-` little endian
`DQ 123456789ABCDEF1h`	`-|12|34|56|78|9A|BC|DE|F1|-` big endian `-|F1|DE|BC|9A|78|56|34|12|-` little endian
`DB '1234'`	`-|31|32|33|34|-` big endian `-|31|32|33|34|-` little endian

Ejercicio

¿Cuál es el rango de representación de los números enteros sin signo con 8, 16 y 32 bits de precisión? ¿Cuál es el rango de representación de los números enteros en complemento a dos con 8, 16 y 32 bits de precisión?

Ejercicio

¿Cuál es el rango de representación de los números enteros sin signo con 8, 16 y 32 bits de precisión? ¿Cuál es el rango de representación de los números enteros en complemento a dos con 8, 16 y 32 bits de precisión?

Sin signo	0 a $2^n - 1$
Con signo	-2^{n-1} a $2^{n-1} - 1$

	Sin signo	Con signo
8	0 a 255	−128 a 127
16	0 a 65535	−32768 a 32767
32	0 a 4294967295	−2147483648 a 2147483647

Ejercicio

Exprese los números 133 y 123 en notación binaria con 8 bits de precisión (notación sin signo), y realice la suma de estos dos números bit a bit. Luego, exprese los números -123 y 123 en notación complemento a dos con 8 bits de precisión y realice la suma de estos dos números bit a bit. ¿Qué conclusión puede sacar al observar el resultados de ambas operaciones?

Ejercicio

Exprese los números 133 y 123 en notación binaria con 8 bits de precisión (notación sin signo), y realice la suma de estos dos números bit a bit. Luego, exprese los números -123 y 123 en notación complemento a dos con 8 bits de precisión y realice la suma de estos dos números bit a bit. ¿Qué conclusión puede sacar al observar el resultados de ambas operaciones?

```
  123 = 01111011
 -123 = 10000101
-----------------
      100000000
```

```
  123 = 01111011
  133 = 10000101
-----------------
      100000000
```

Esa es la razón por la cual no hay dos ADD/SUB,
sino uno solo tanto para números con signo como sin signo.

Es responsabilidad del programador saber con qué tipo de números se está operando, y prestar atención a los flags correctos.

Ejercicio

Explique qué indican y cuándo se setean los flags de paridad (PF), de cero (ZF) y de signo (SF). Explique las diferencias entre el flag de carry (CF) y el flag de overflow (OF).

Importante: Los flags se setean dependiendo de la operación. La interpretación depende del programador.

Ejercicio

Explique qué indican y cuándo se setean los flags de paridad (PF), de cero (ZF) y de signo (SF). Explique las diferencias entre el flag de carry (CF) y el flag de overflow (OF).
Importante: Los flags se setean dependiendo de la operación. La interpretación depende del programador.

$CF = 1$	Bit más significativo en la suma. En la resta si hay *borrow*.
$CF = 0$	cualquier otro caso
$OF = 1$	Si hay overflow (el resultado esta fuera de la representación)
$OF = 0$	cualquier otro caso
$PF = 1$	Si el byte menos significativo tiene un número par de 1s
$PF = 0$	cualquier otro caso
$SF = 1$	Si el bit más significativo es 1
$SF = 0$	cualquer otro caso
$ZF = 1$	Si el resultado es cero
$ZF = 0$	cualquier otro caso

Ejercicio

Indique cuáles son las condiciones para que se activen las siguientes instrucciones de salto: JA, JAE, JB, JBE, JE, JG, JGE, JL, JLE y JZ.

Ejercicio

Indique cuáles son las condiciones para que se activen las siguientes instrucciones de salto: JA, JAE, JB, JBE, JE, JG, JGE, JL, JLE y JZ.

JA	CF=0 and ZF=0	Above
JAE	CF=0	Above or Equal
JB	CF=1	Below
JBE	CF=1 or ZF=1	Below or Equal
JE	ZF=1	Equal
JG	ZF=0 and SF=OF	Greater (signed)
JGE	SF=OF	Greater or Equal (signed)
JL	SF != OF	Less (signed)
JLE	ZF=1 or SF != OF	Less or Equal (signed)
JZ	ZF=1	Zero

Instrucciones y Registros

Operaciones

`ADD`, `SUB`, `MOV`, `SHL`, `JMP` ...

Instrucciones y Registros

Operaciones

`ADD`, `SUB`, `MOV`, `SHL`, `JMP` ... (ver manual)

Instrucciones y Registros

Operaciones

`ADD`, `SUB`, `MOV`, `SHL`, `JMP` ... (ver manual)

Registros

8 bits: `AL BL CL DL DIL SIL BPL SPL R8B ... R15B`

16 bits: `AX BX CX DX DI SI BP SP R8W ... R15W`

32 bits: `EAX EBX ECX EDX EDI ESI EBP ESP R8D ... R15D`

64 bits: `RAX RBX RCX RDX RSI RDI RBP RSP R8 ... R15`

128 bits: `XMMO, ... , XMM15`

Instrucciones y Registros

Operaciones

`ADD`, `SUB`, `MOV`, `SHL`, `JMP` ... (ver manual)

Registros

8 bits:	AL	BL	CL	DL	DIL	SIL	BPL	SPL	R8B	...	R15B
16 bits:	AX	BX	CX	DX	DI	SI	BP	SP	R8W	...	R15W
32 bits:	EAX	EBX	ECX	EDX	EDI	ESI	EBP	ESP	R8D	...	R15D
64 bits:	RAX	RBX	RCX	RDX	RSI	RDI	RBP	RSP	R8	...	R15

128 bits: XMM0, ... , XMM15

Direccionamiento

[Base + (Index * Scale) + Displacement]

Base	Index	Scale	Displacement
RAX	RAX	1	Cte. 32 bits
...	...	2	
R15	R15	4	
	(NO RSP)	8	

Hola Mundo ...

Ejercicio

Escriba un programa en lenguaje ensamblador que imprima por pantalla:

```
Hola Mundo
```

Hola Mundo ...

Ejercicio

Escriba un programa en lenguaje ensamblador que imprima por pantalla:

```
Hola Mundo
```

¿Cómo?

Secciones, etiquetas y símbolos

Un programa assembler se separa en secciones

- `.data`: Donde declarar variables globales inicializadas. (`DB`, `DW`, `DD` y `DQ`).
- `.rodata`: Donde declarar constantes globales inicializadas. (`DB`, `DW`, `DD` y `DQ`).
- `.bss`: Donde declarar variables globales no inicializadas. (`RESB`, `RESW`, `RESD` y `RESQ`).
- `.text`: Es donde se escribe el código.

Secciones, etiquetas y símbolos

Un programa assembler se separa en secciones

- `.data`: Donde declarar variables globales inicializadas. (`DB`, `DW`, `DD` y `DQ`).
- `.rodata`: Donde declarar constantes globales inicializadas. (`DB`, `DW`, `DD` y `DQ`).
- `.bss`: Donde declarar variables globales no inicializadas. (`RESB`, `RESW`, `RESD` y `RESQ`).
- `.text`: Es donde se escribe el código.

Etiquetas y símbolos

- `global`: Define un símbolo que va a ser visto externamente
- `_start`: Punto de entrada de un programa en linux

Pseudoinstrucciones

Son instrucciones para el ensamblador

- `DB`, `DW`, `DD`, `DQ`, `RESB`, `RESW`, `RESD` y `RESQ`.
- expresión `$`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión.

Pseudoinstrucciones

Son instrucciones para el ensamblador

- `DB`, `DW`, `DD`, `DQ`, `RESB`, `RESW`, `RESD` y `RESQ`.
- expresión `$`, se evalúa en la posición en memoria al principio de la línea que contiene la expresión.
- comando `EQU`, para definir constantes que después no quedan en el archivo objeto.
- comando `INCBIN`, incluye un binario en un archivo assembler.
- prefijo `TIMES`, repite una cantidad de veces la instrucción que le sigue.

Llamadas al sistema operativo (syscalls)

Utilizando la famosa `int 0x80` (en Linux) solicitamos al Sistema Operativo que haga algo por nosotros.

Su interfaz es:

1- El número de función que queremos en rax

2- Los parámetros en rbx, rcx, rdx, rsi, rdi y rbp; en ese orden

3- Llamamos a la interrupción del sistema operativo (int 0x80)

4- En general, la respuesta está en rax

Llamadas al sistema operativo (syscalls)

Utilizando la famosa `int 0x80` (en Linux) solicitamos al Sistema Operativo que haga algo por nosotros.

Su interfaz es:

1- El número de función que queremos en rax

2- Los parámetros en rbx, rcx, rdx, rsi, rdi y rbp; en ese orden

3- Llamamos a la interrupción del sistema operativo (int 0x80)

4- En general, la respuesta está en rax

- Mostrar por pantalla (**sys_write**):
  Función **4**
  Parámetro 1: **¿donde?** (1 = stdout)
  Parámetro 2: **Dirección de memoria del mensaje**
  Parámetro 3: **Longitud del mensaje** (en bytes)
- Terminar programa (**exit**):
  Función **1**
  Parámetro 1: **código de retorno** (0 = sin error)

Hola Mundo... solución

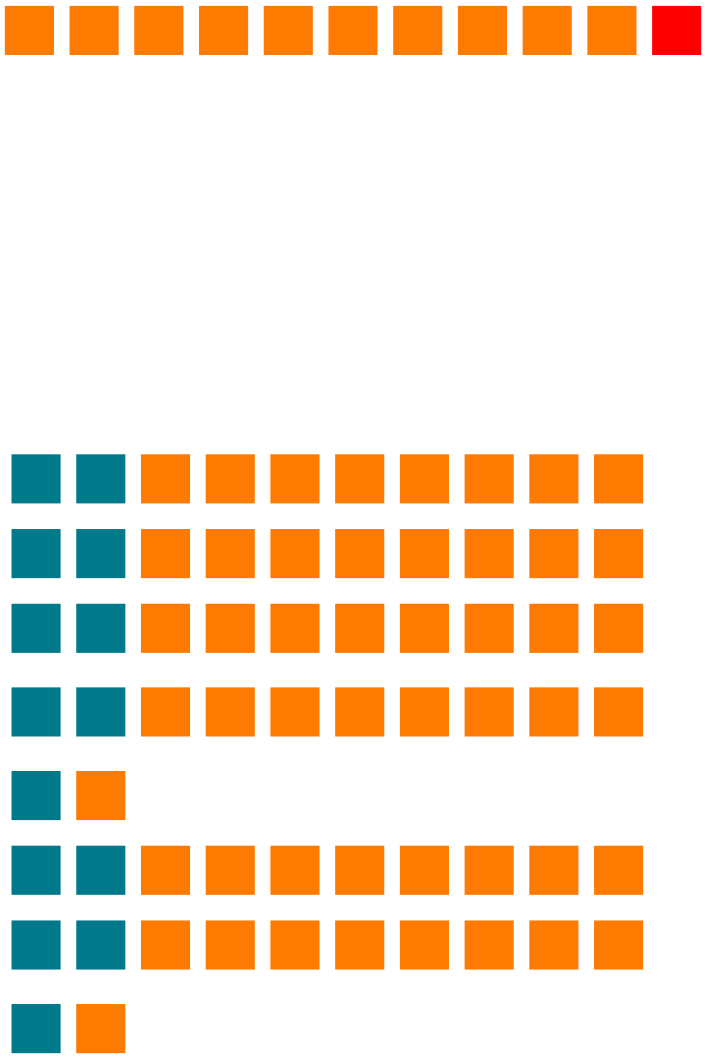
```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text
  _start:
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo  ; longitud
    int 0x80
    mov rax, 1      ; funcion 1
    mov rbx, 0      ; codigo
    int 0x80
```

Hola Mundo... solución

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text
  _start:
    mov rax, 4       ; funcion 4
    mov rbx, 1       ; stdout
    mov rcx, msg     ; mensaje
    mov rdx, largo   ; longitud
    int 0x80
    mov rax, 1       ; funcion 1
    mov rbx, 0       ; codigo
    int 0x80
```



Ensamblando y linkeando

Ensamblamos:

```
nasm -f elf64 holamundo.asm
```

Linkeamos:

```
ld -o holamundo holamundo.o
```

Ejecutamos:

```
./holamundo
```

Debugger

Comandos Básicos

`r | run`	Ejecuta el programa hasta el primer break
`b | break FILE:LINE`	Breakpoint en la línea
`b | break FUNCTION`	Breakpoint en la función
`info breakpoints`	Muestra información sobre los breakpoints
`c | continue`	Continúa con la ejecución
`s | step`	Siguiente línea (Into)
`n | next`	Siguiente línea (Over)
`si | stepi`	Siguiente instrucción asm (Into)
`ni | nexti`	Siguiente instrucción asm (Over)
`x/Nuf ADDR`	Muestra los datos en memoria

`N` = Cantidad (bytes)

`u` = Unidad `b|h|w|g`

`b:byte, h:word, w:dword, g:qword`

`f` = Formato `x|d|u|o|f|a`

`x`:hex, `d`:decimal, `u`:decimal sin signo, `o`:octal, `f`:float, `a`:direcciones

Configuración de GDB:

```
~/.gdbinit
```

Para usar sintaxis intel y guardar historial de comandos:

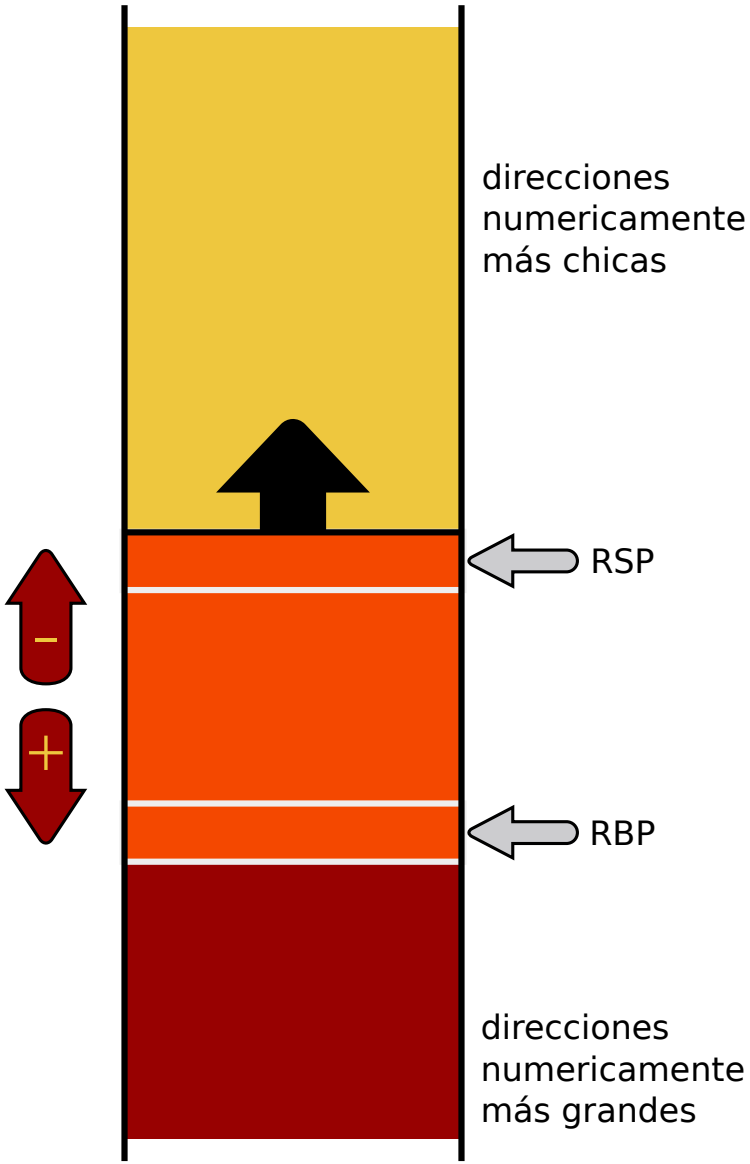
```
set disassembly-flavor intel
set history save
```

Correr GDB con argumentos:

```
gdb --args <ejecutable> <arg1> <arg2> ...
```

Pila y Convención C

Pila - Estructura



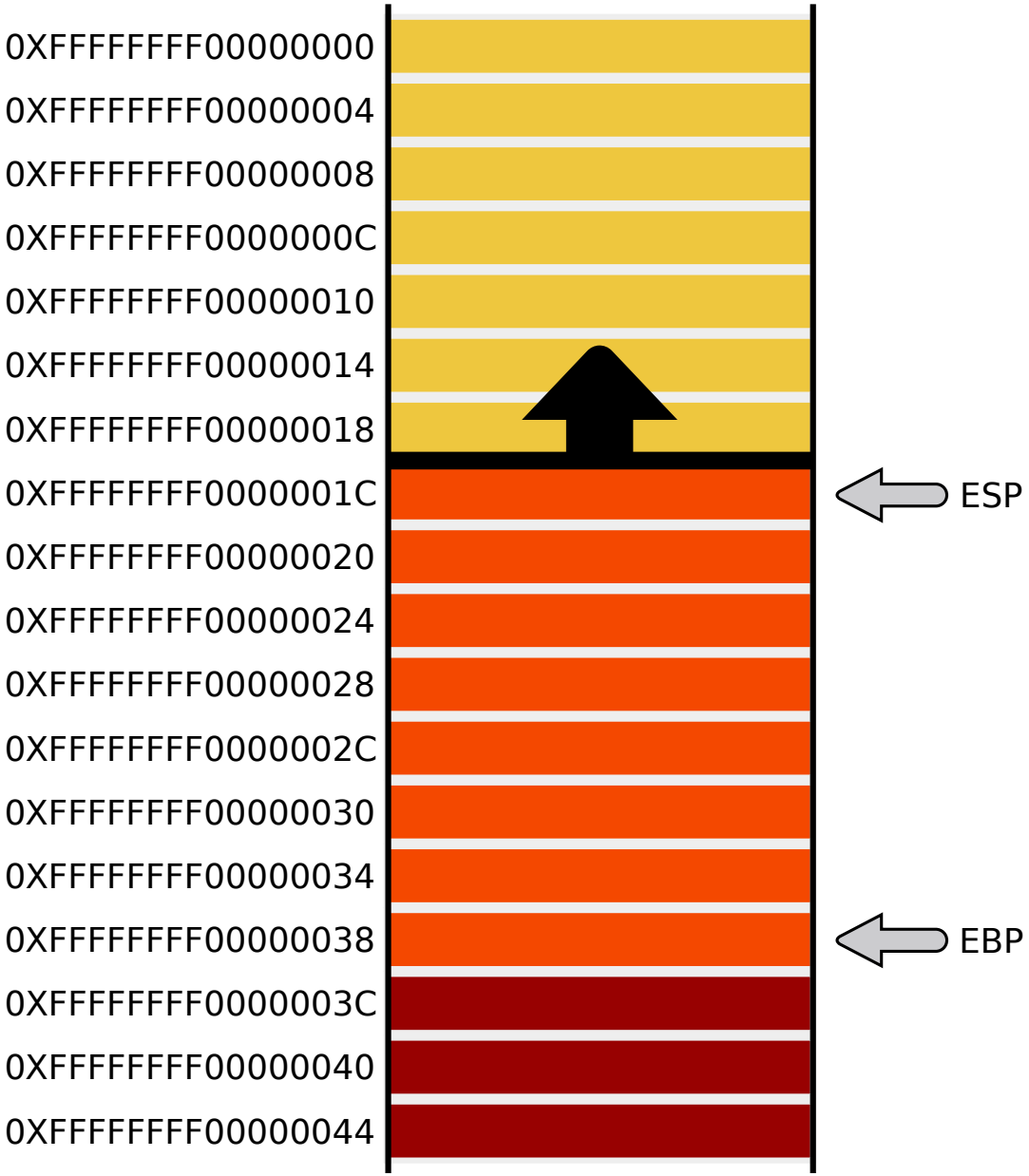
En 32 bits

- 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)

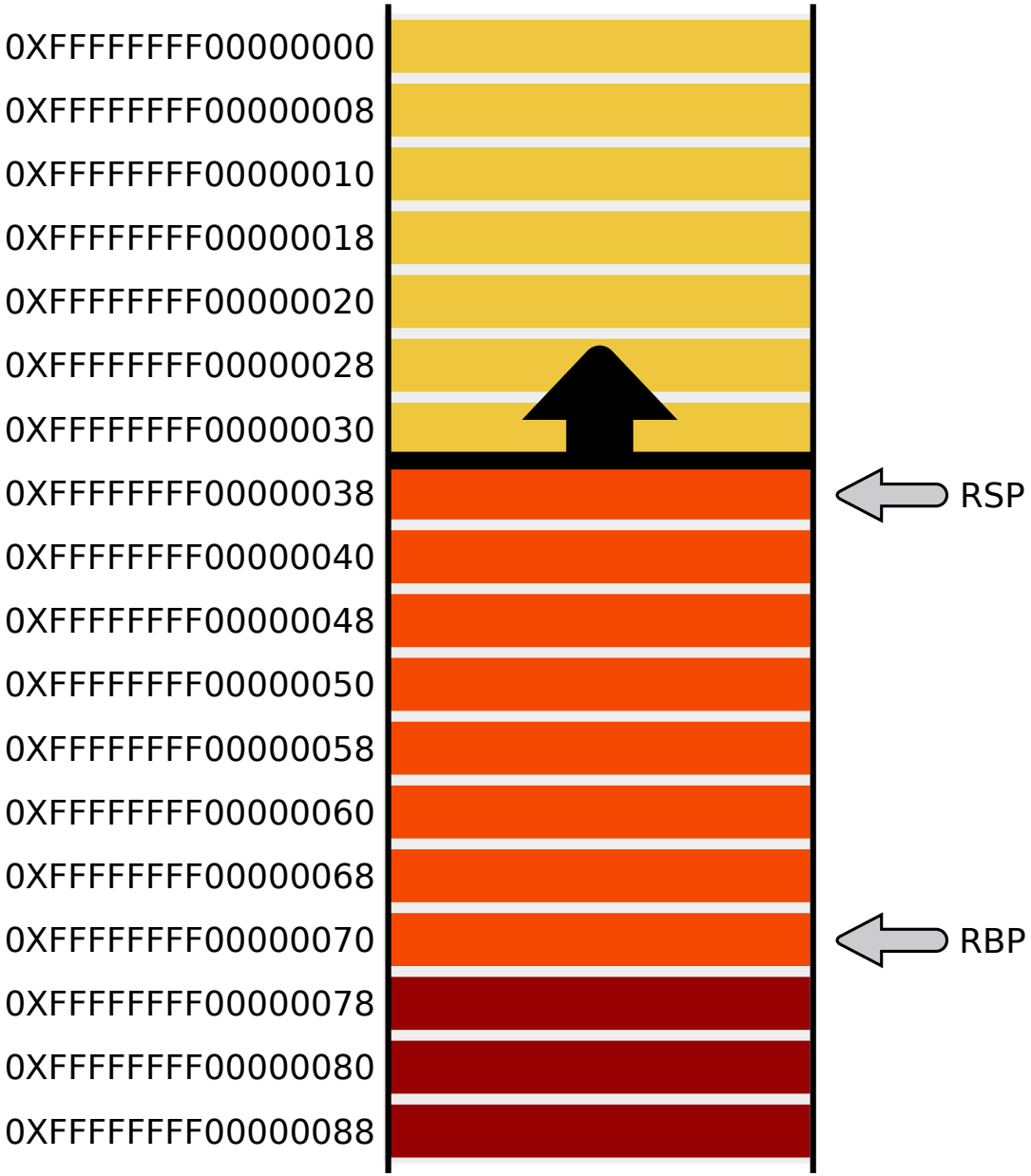
Instrucciones

- PUSH
- POP

Registros

- ESP
- EBP

Pila en 64 bits - Estructura



Alineación

8 bytes (en 64bits)

16 bytes para llamar funciones

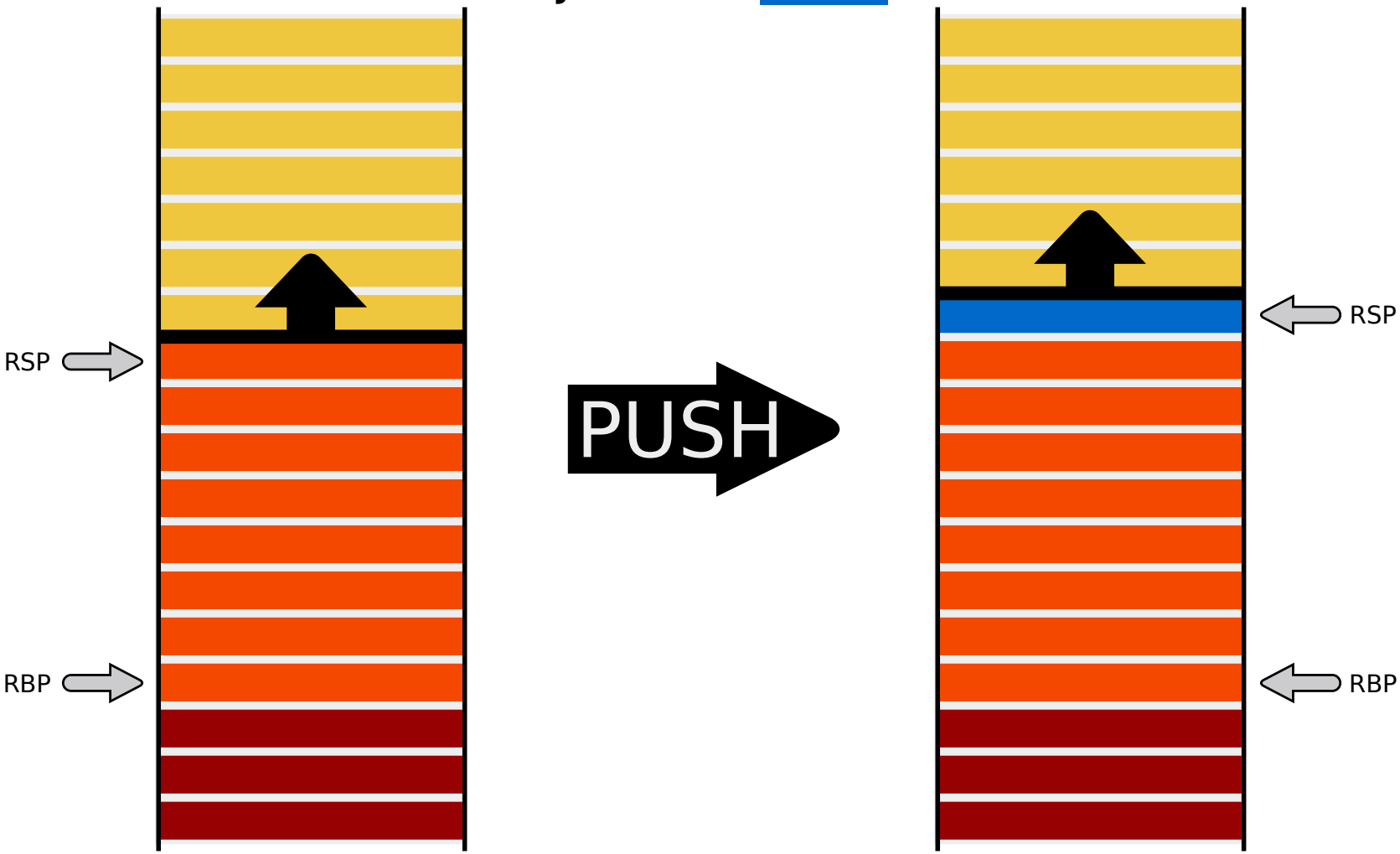
Instrucciones

- PUSH
- POP

Registros

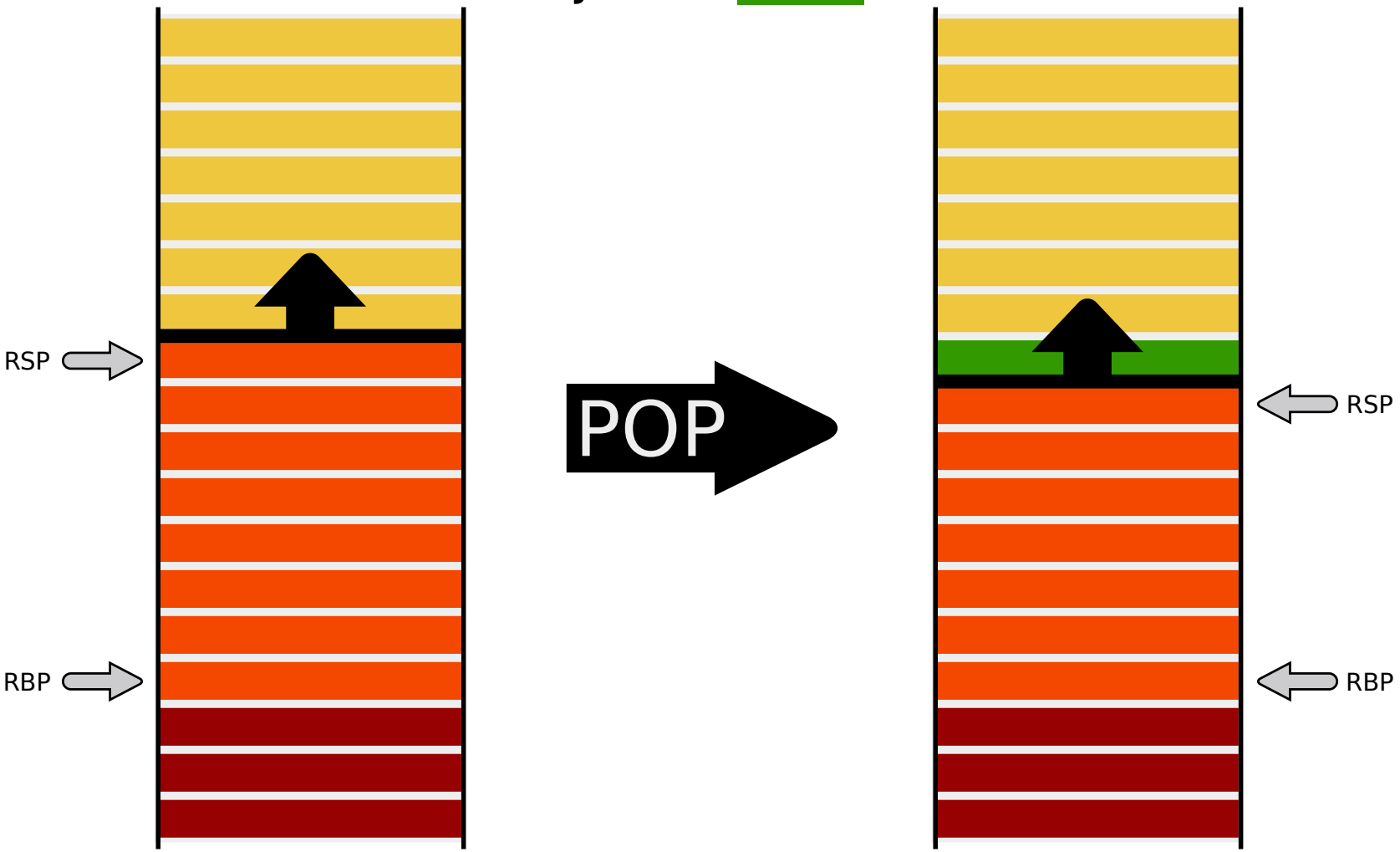
- RSP
- RBP

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.

- 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.
  - Cómo las funciones retornan el resultado.
  - Qué registros se deben preservar en una función.

- Las convenciones dependen de la arquitectura del procesador:
  - En `x86` (32bits) se conoce como `x32 ABI`.
  - En `x86-64` (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.

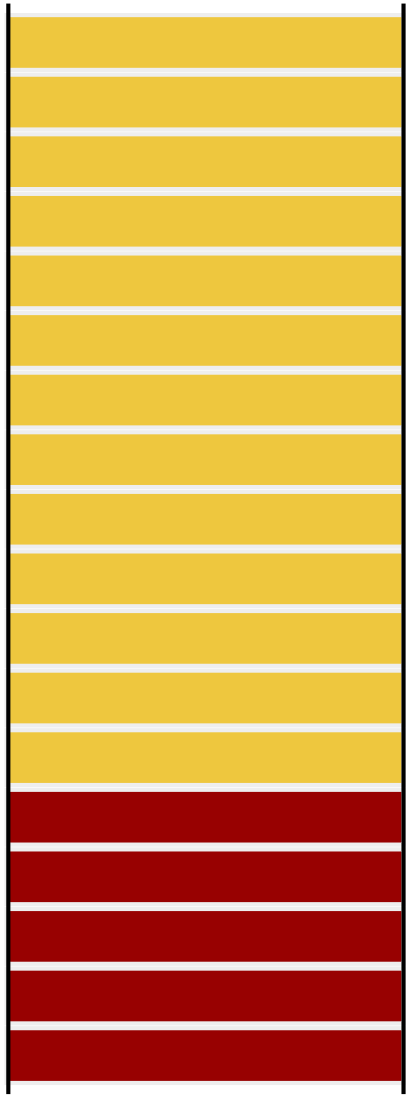
La construcción del *stack frame* consiste en colocar el registro base de la pila en una dirección relativa al comienzo del área de la función llamadora.

Caso 32 bits

- Preservar los registros `EBX`, `ESI`, `EDI` y `ESP`.
- Retornar el resultado a través de `EAX` (y `EDX` si ocupa 64b).
- Preservar la consistencia de la pila.
- Los parametros 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

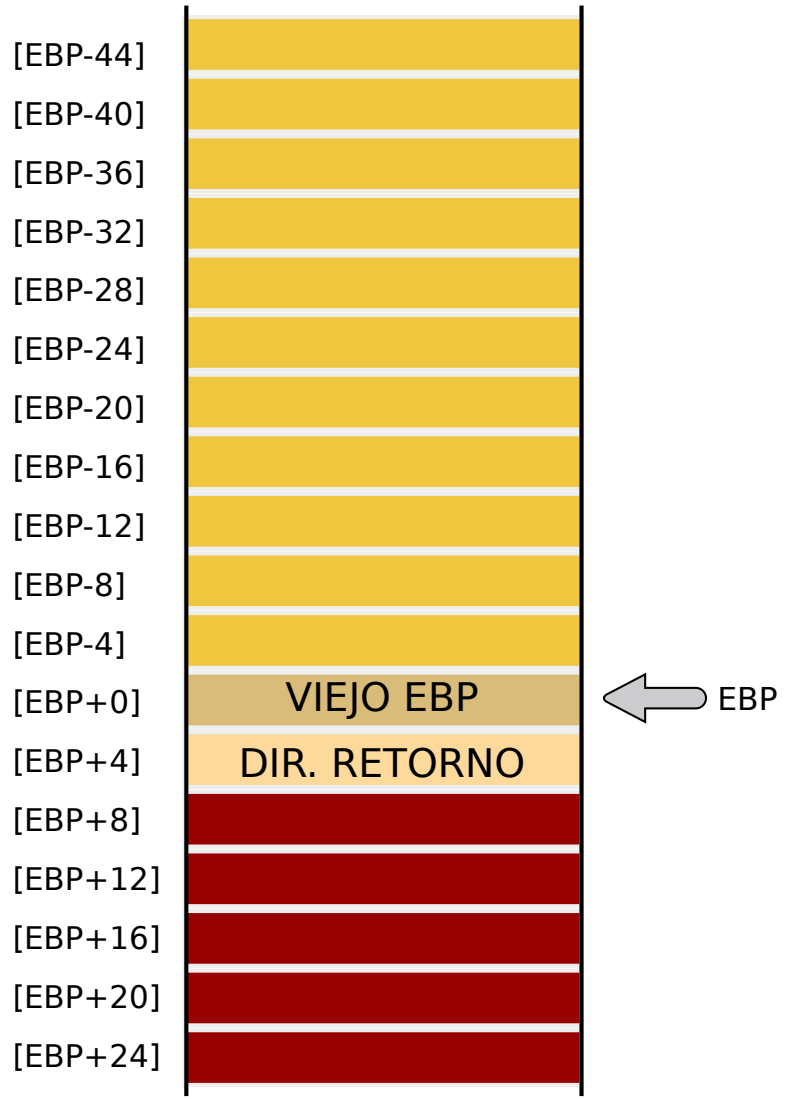
Caso 32 bits

fun:

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

Caso 32 bits

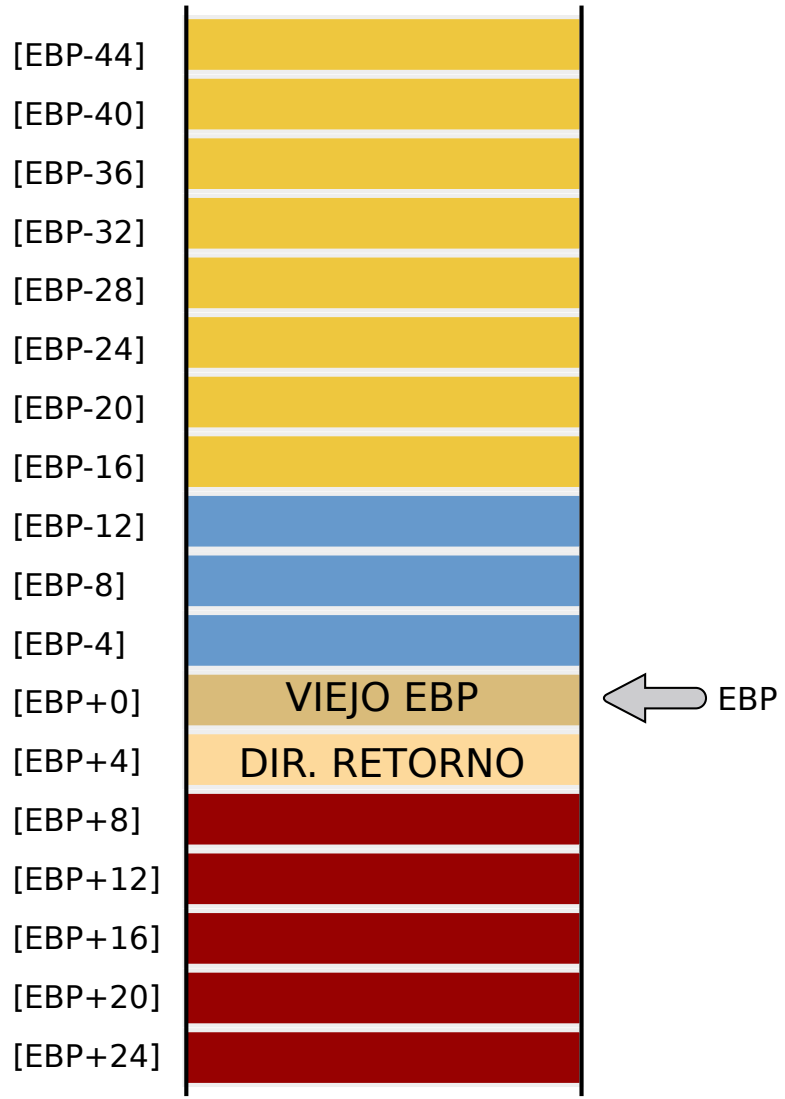
```
fun:
        PUSH EBP
        MOV  EBP,ESP
```



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

Caso 32 bits

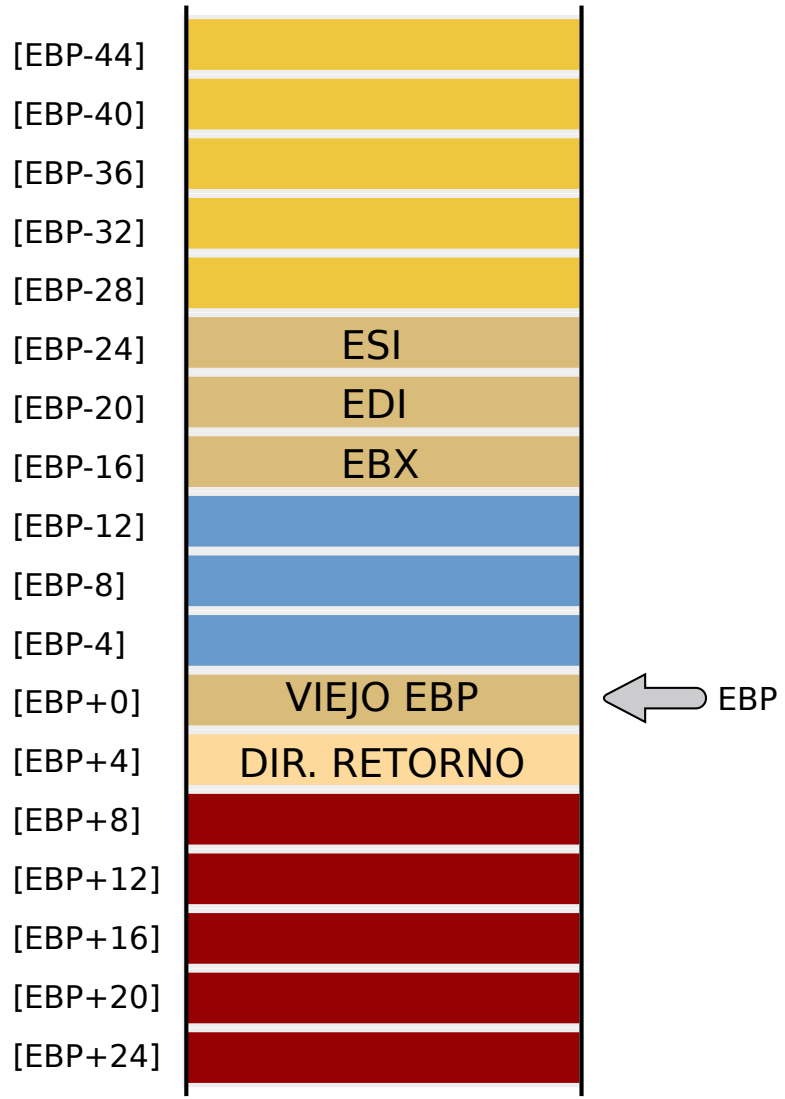
```
fun:
        PUSH EBP
        MOV  EBP,ESP
        SUB ESP,12
```



- 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
```

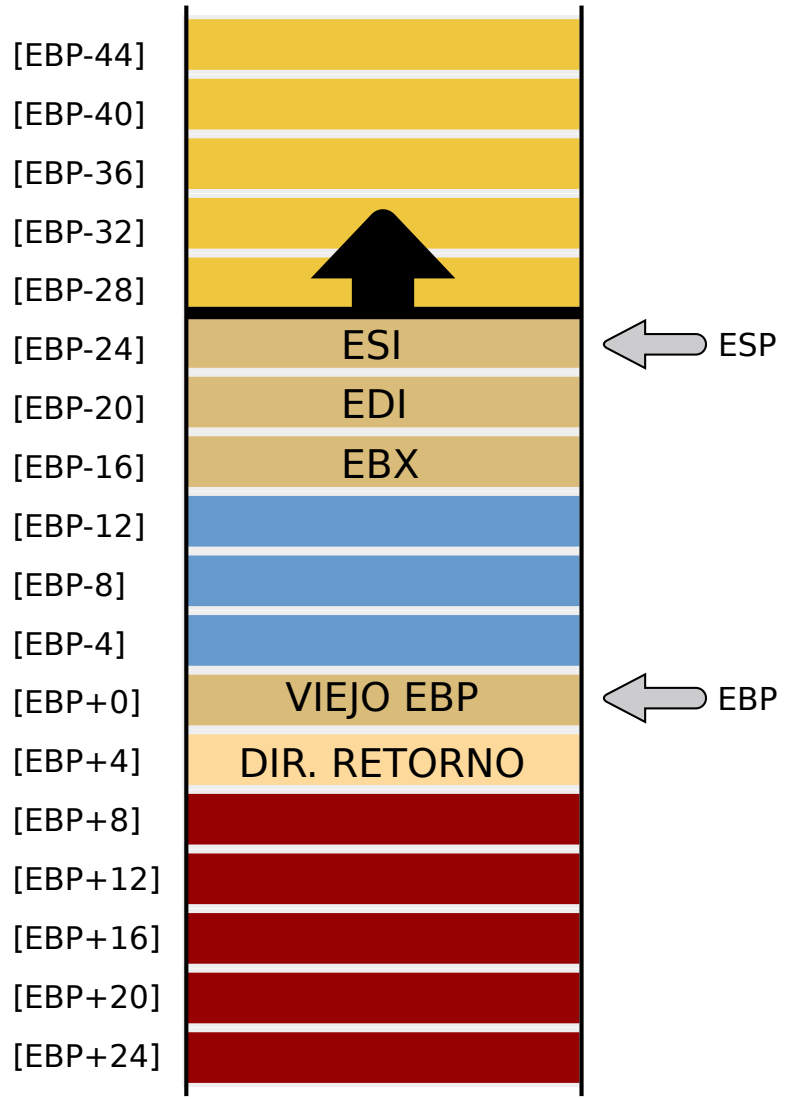


- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 32 bits

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

MI CÓDIGO



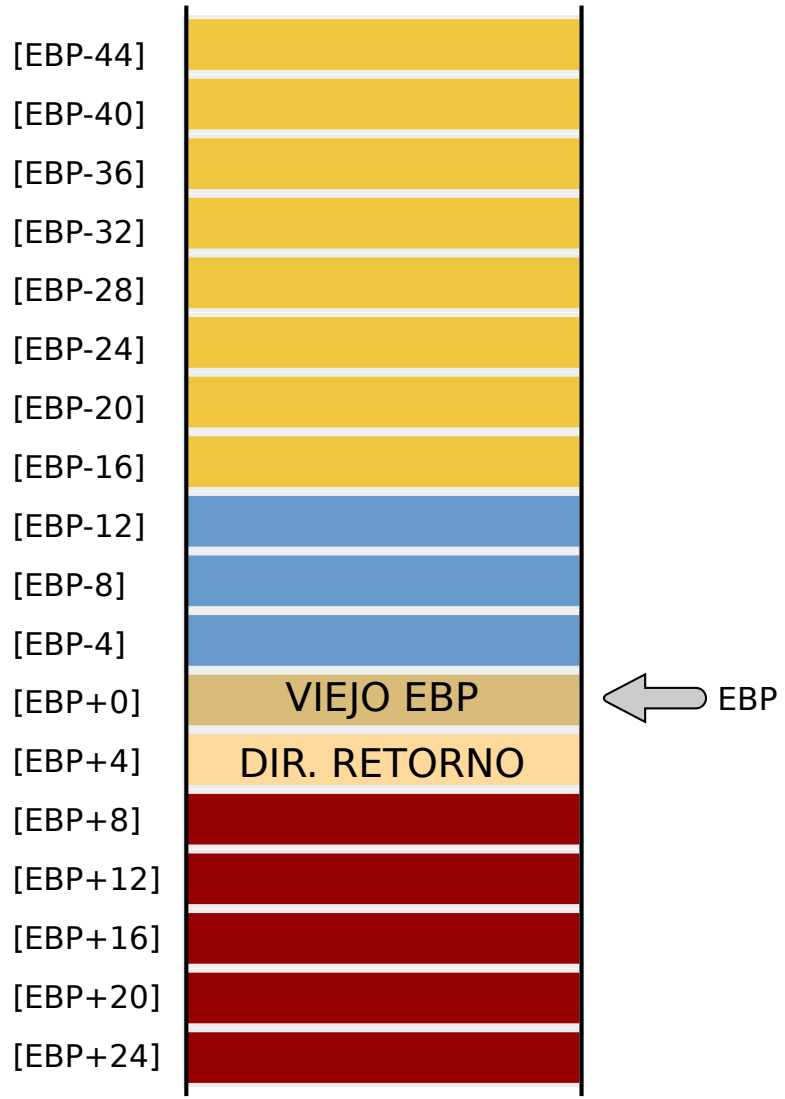
- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

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  RBX
```



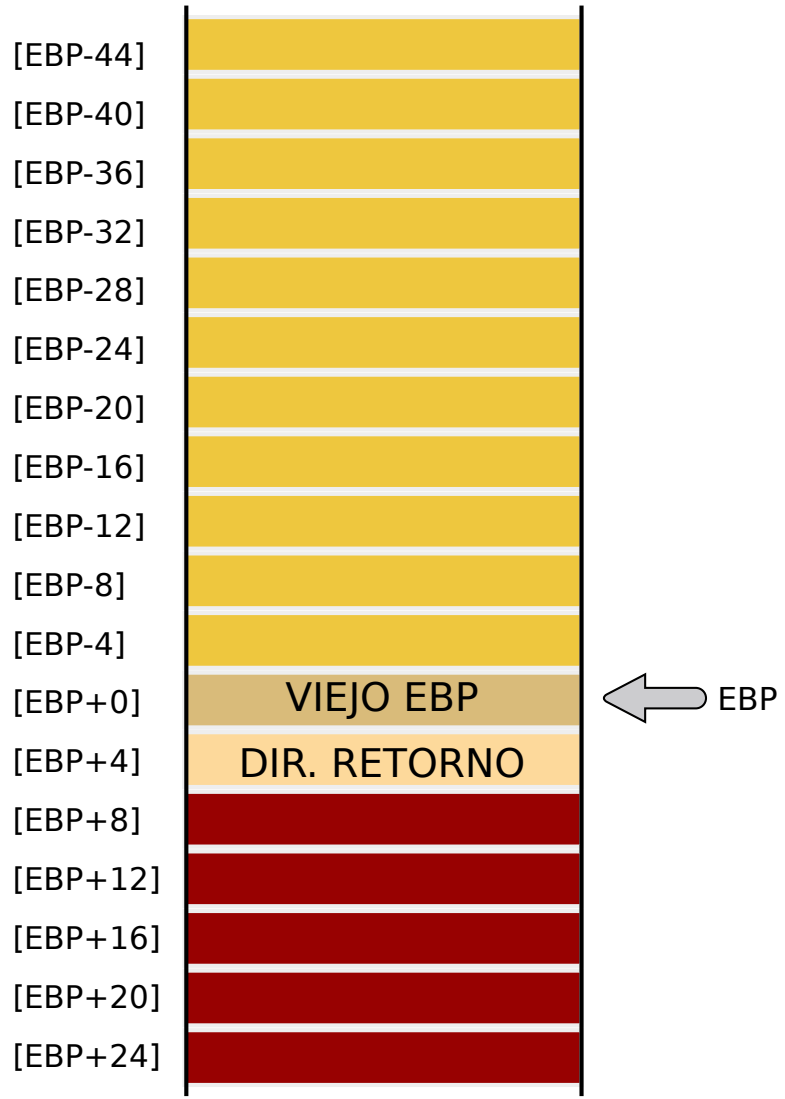
- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

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  RBX
        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  RBX
        ADD ESP,12
        POP  RBP
```

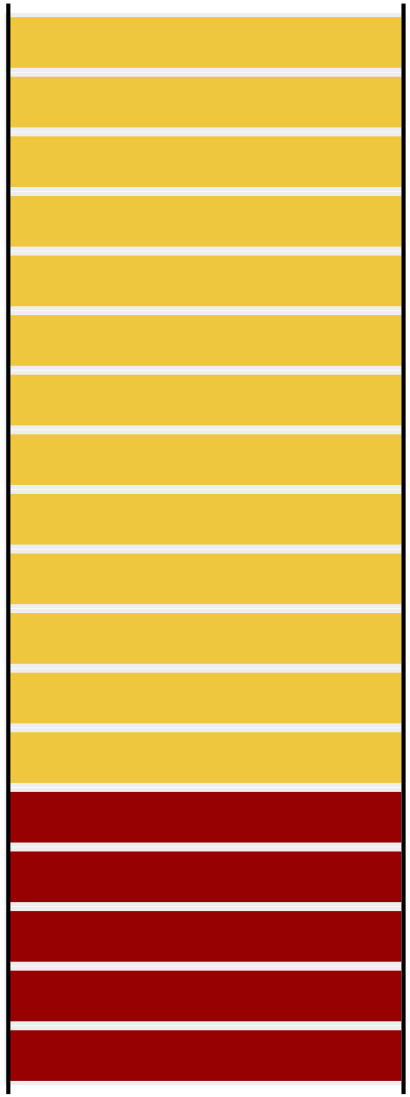
- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

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  RBX
        ADD ESP,12
        POP  RBP
        RET
```



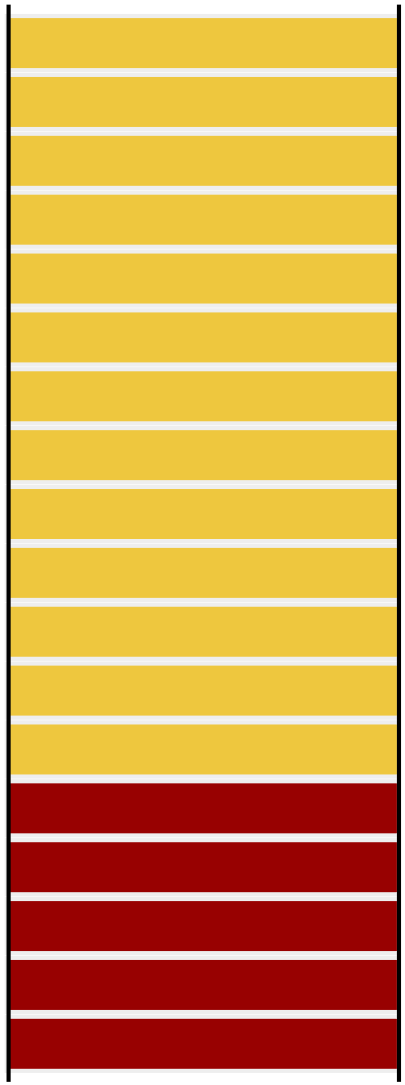
- se guardan los registros EBX, ESI Y EDI
- retorno en EAX

Caso 64 bits

- Preservar los registros `RBX`, `R12`, `R13`, `R14`, `R15` y `RBP`.
- Retornar el resultado a través de `RAX` si un valor entero (y `RDX` 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

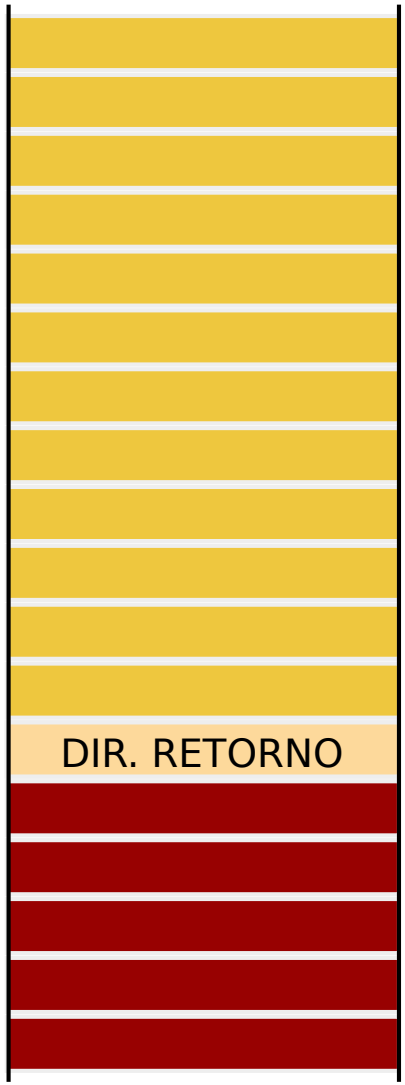
fun:



- 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

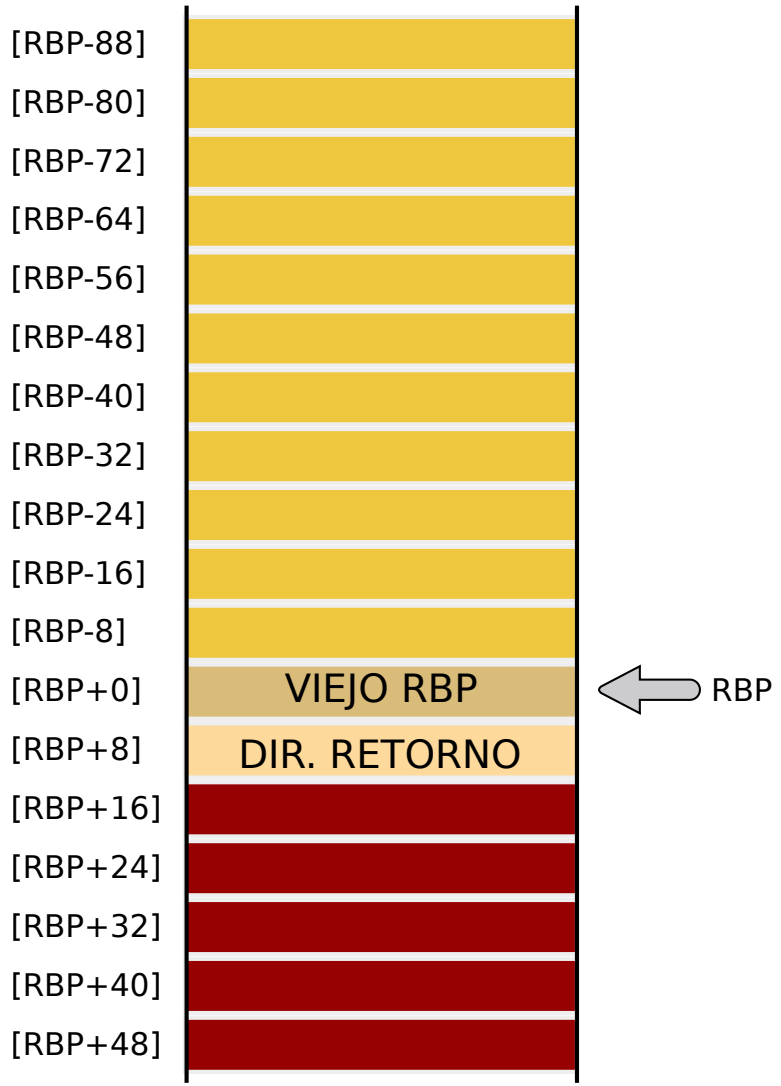
Caso 64 bits

```
fun:
        PUSH RBP
```

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

Caso 64 bits

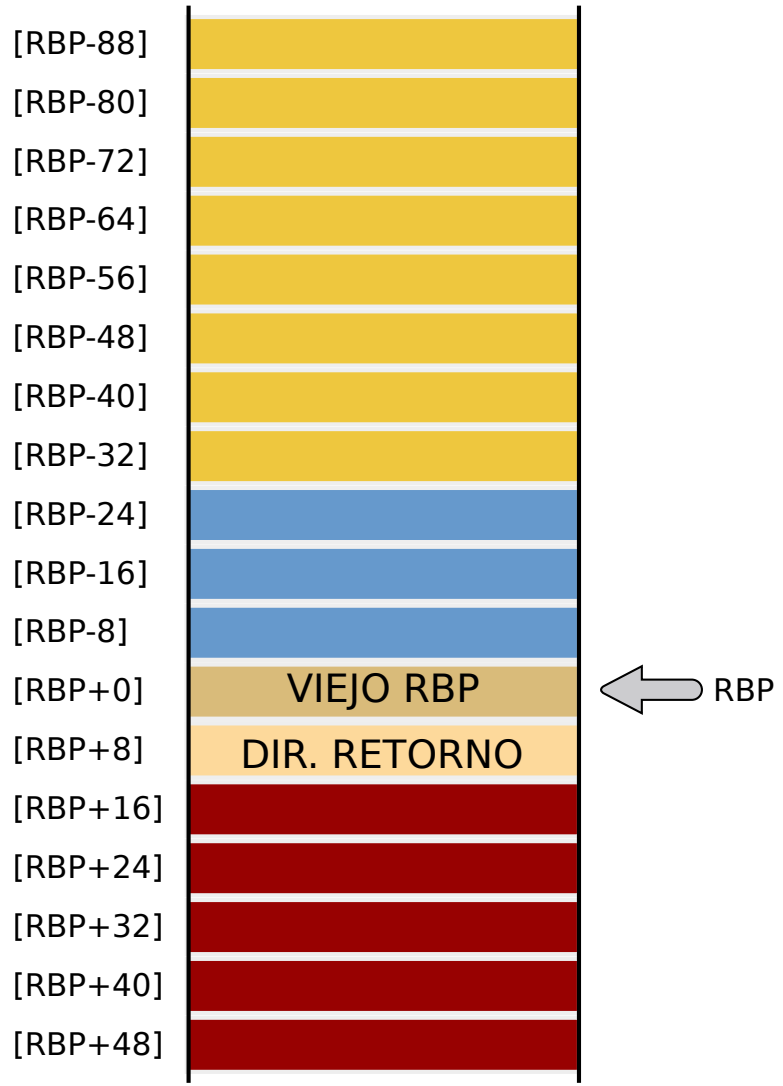
```
fun:
        PUSH RBP
        MOV  RBP,RSP
```



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

Caso 64 bits

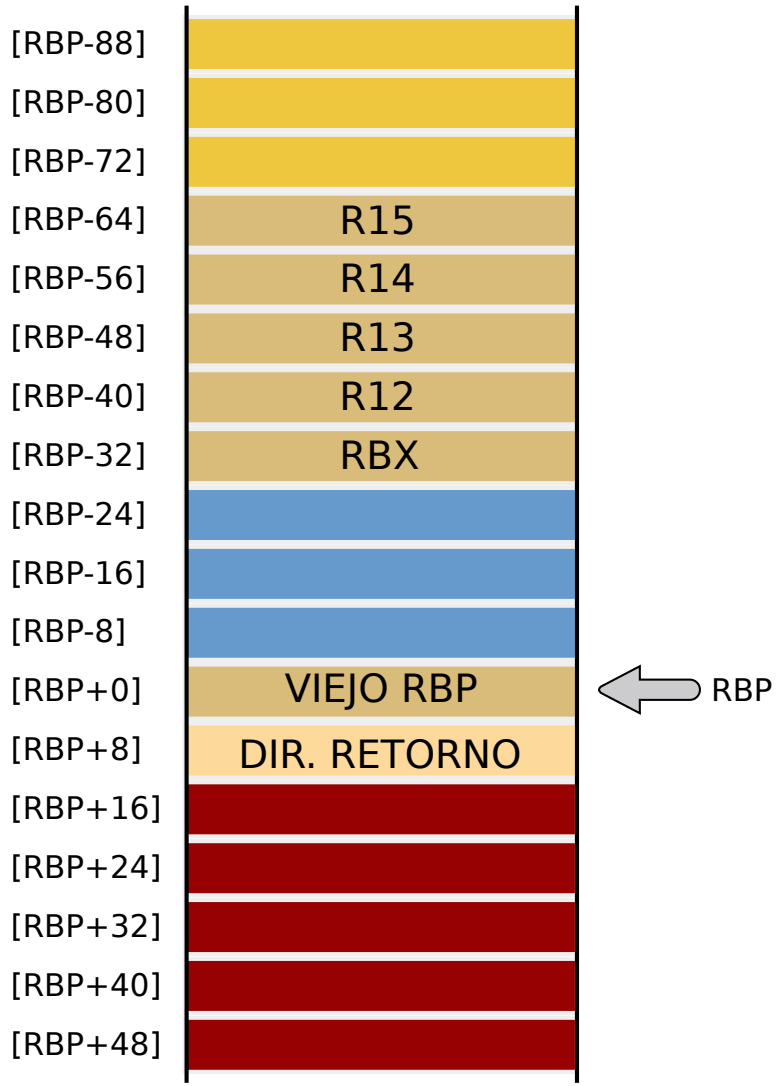
```
fun:
        PUSH RBP
        MOV  RBP,RSP
        SUB RSP,24
```



- 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
```

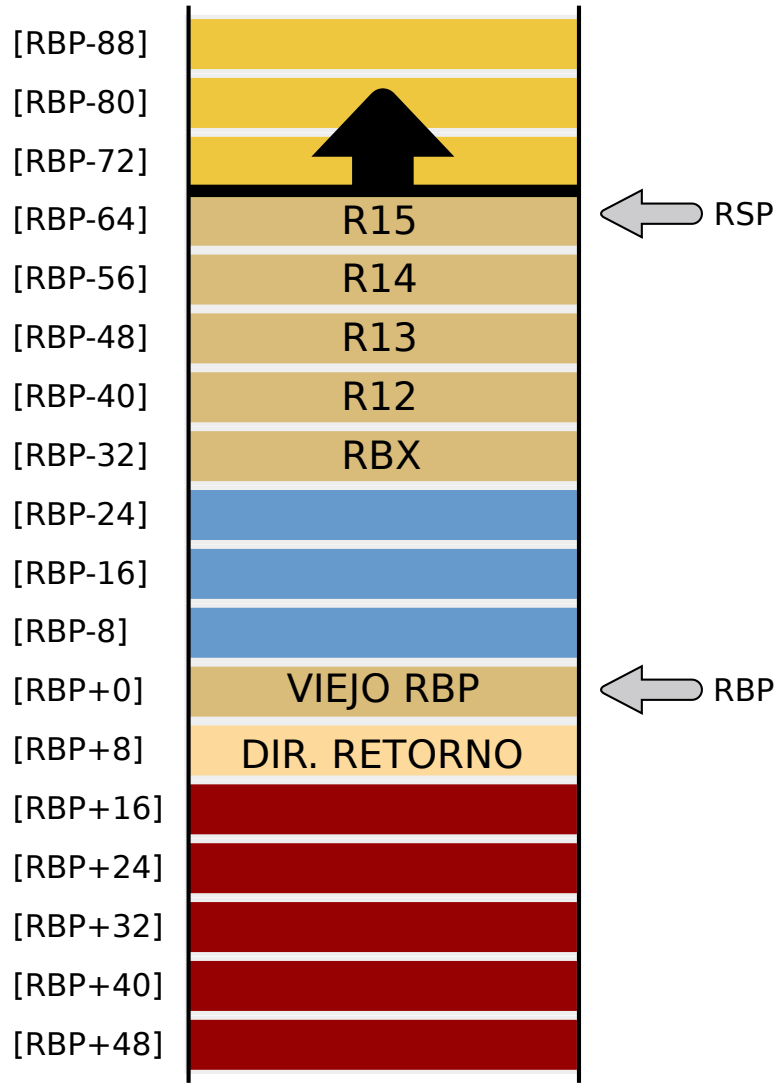


- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

Caso 64 bits

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

MI CÓDIGO



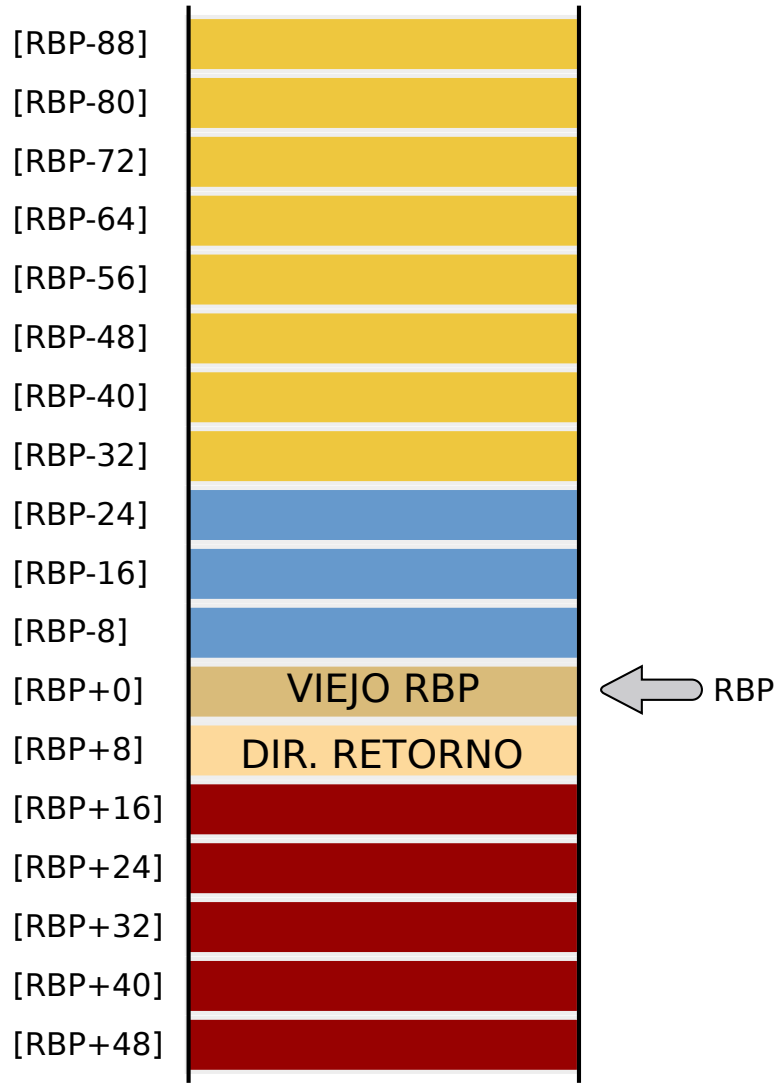
- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

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
```



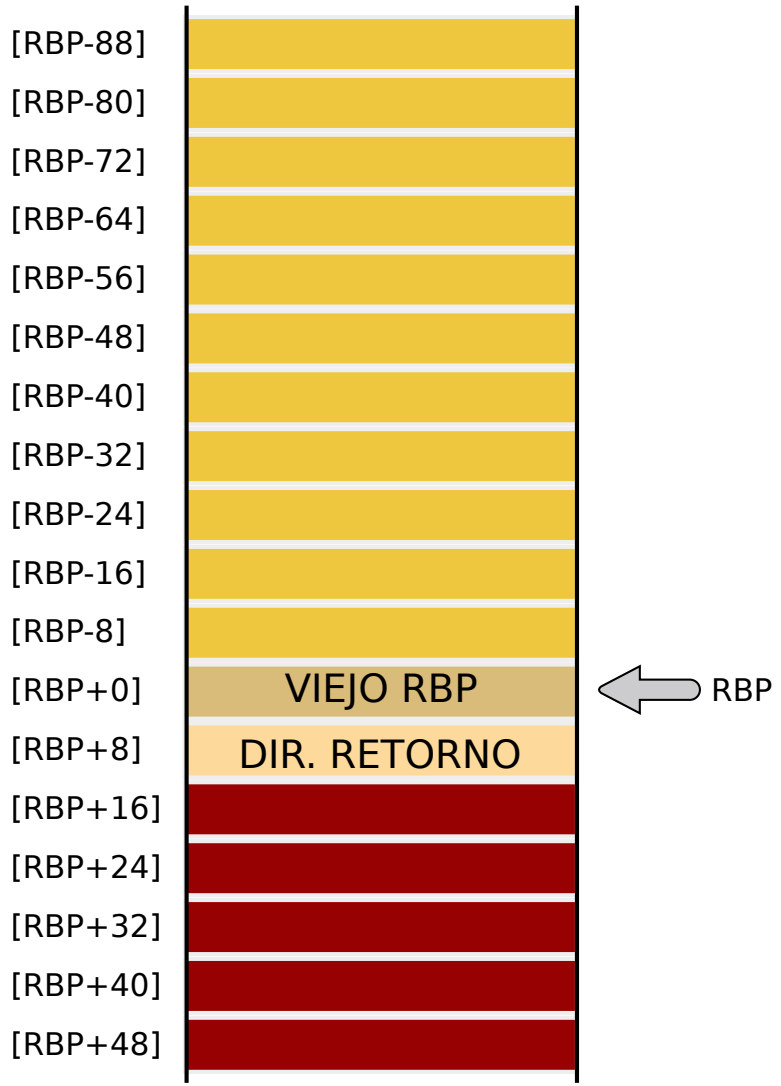
- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

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
```



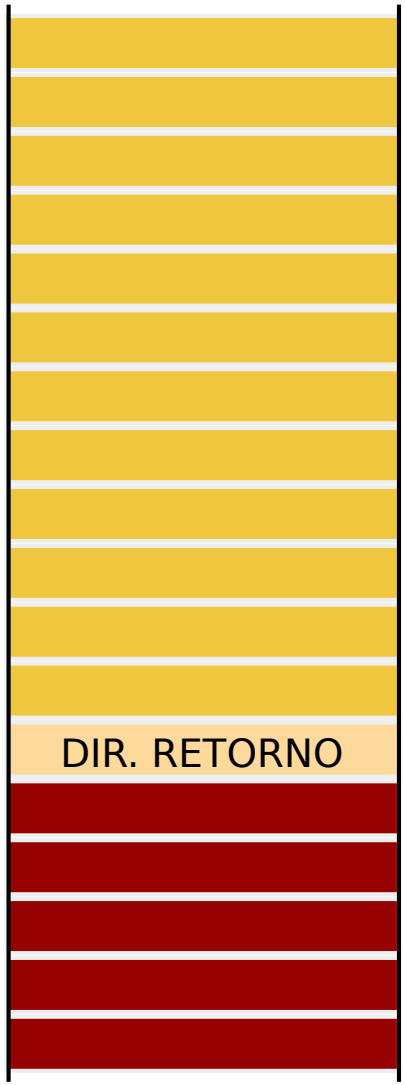
- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

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
```



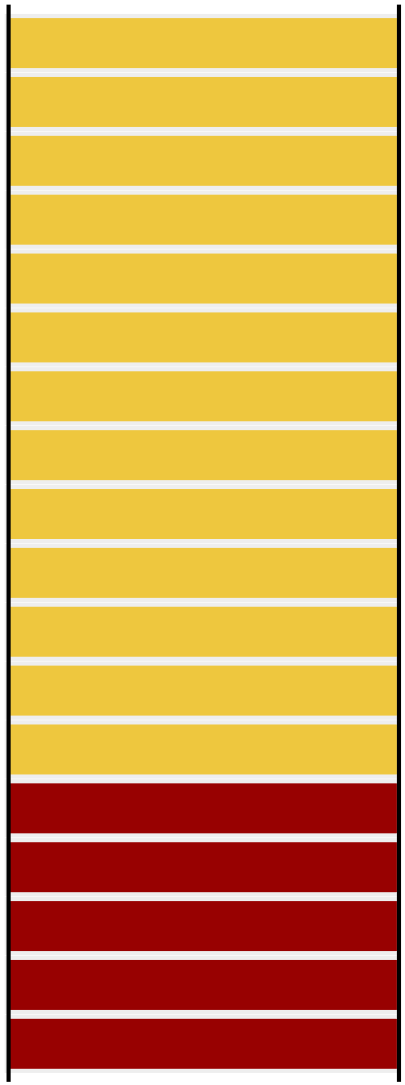
- se guardan los registros RBX, R12, R13, R14 y R15
- retorno en RAX o XMM0

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.
- 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

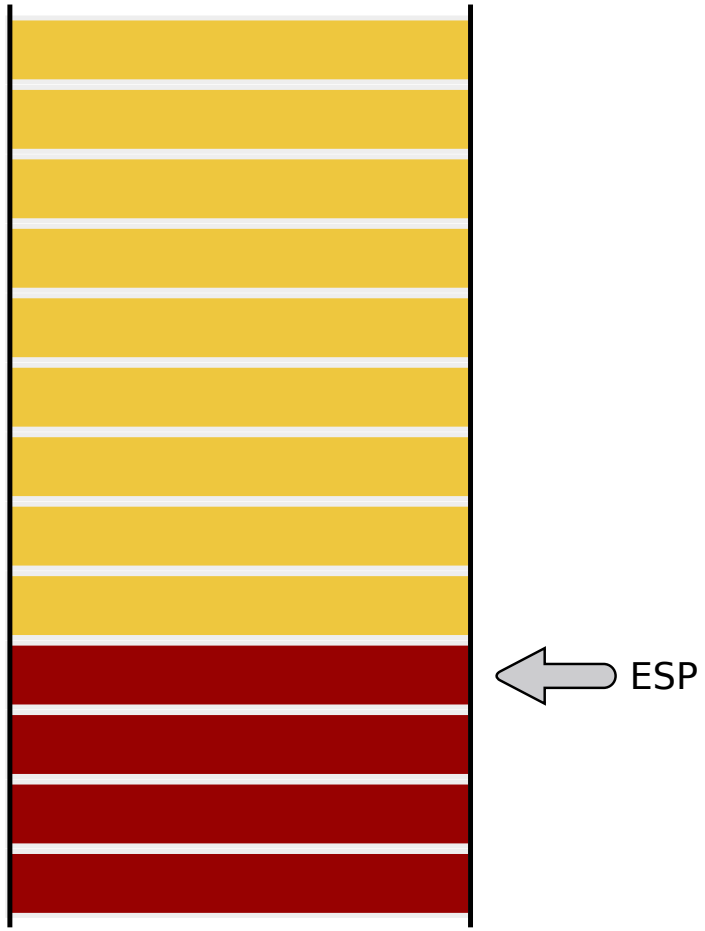
- Los parámetros se pasan por registro, de izquierda a derecha según la firma de la función, clasificados por tipo:
  - Enteros y direcciónes de memoria: `RDI`, `RSI`, `RDX`, `RCX`, `R8` y `R9`
  - Punto flotante: `XMM0` a `XMM7`
  - Resto de los parámetros que superen la cantidad de registros se ubican 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 c2
push b
push a
call f1
add esp, 6*4
...
```

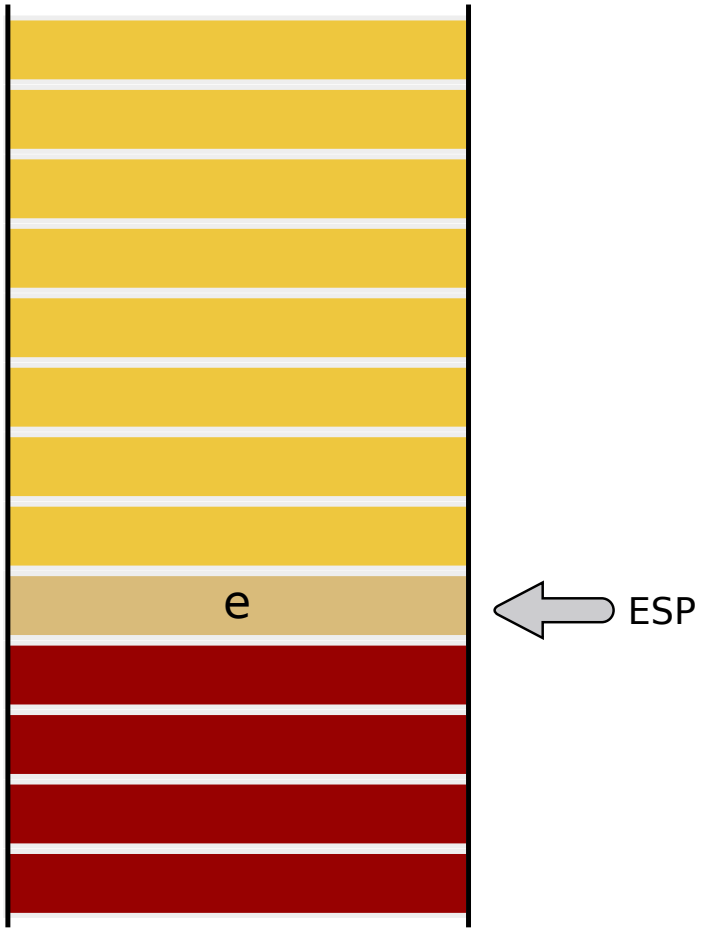


Ejemplo en 32 bits

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

llamado...

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

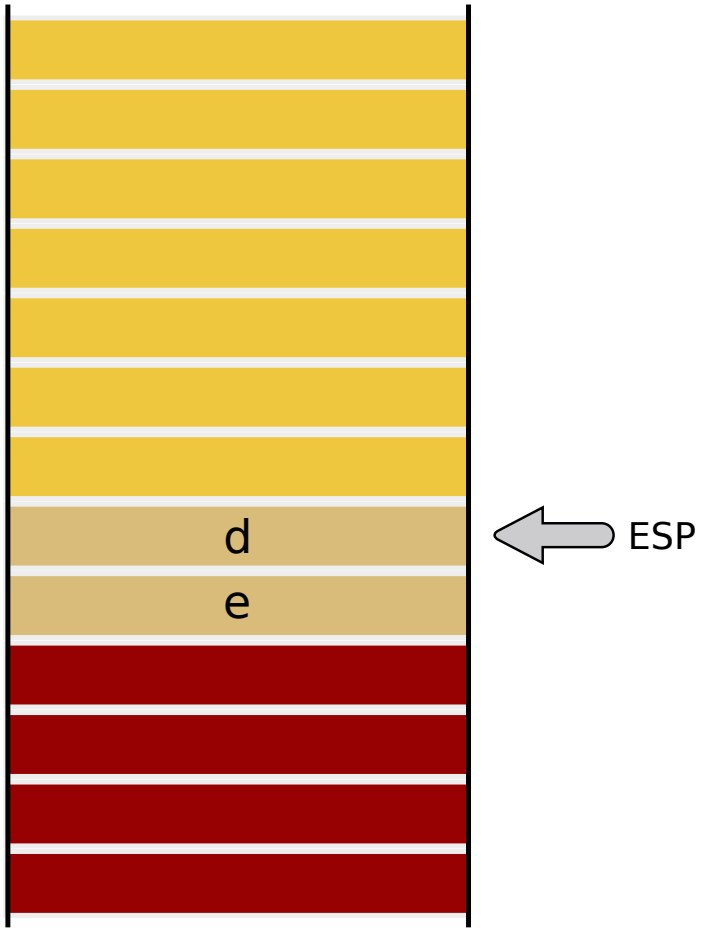


Ejemplo en 32 bits

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

llamado...

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

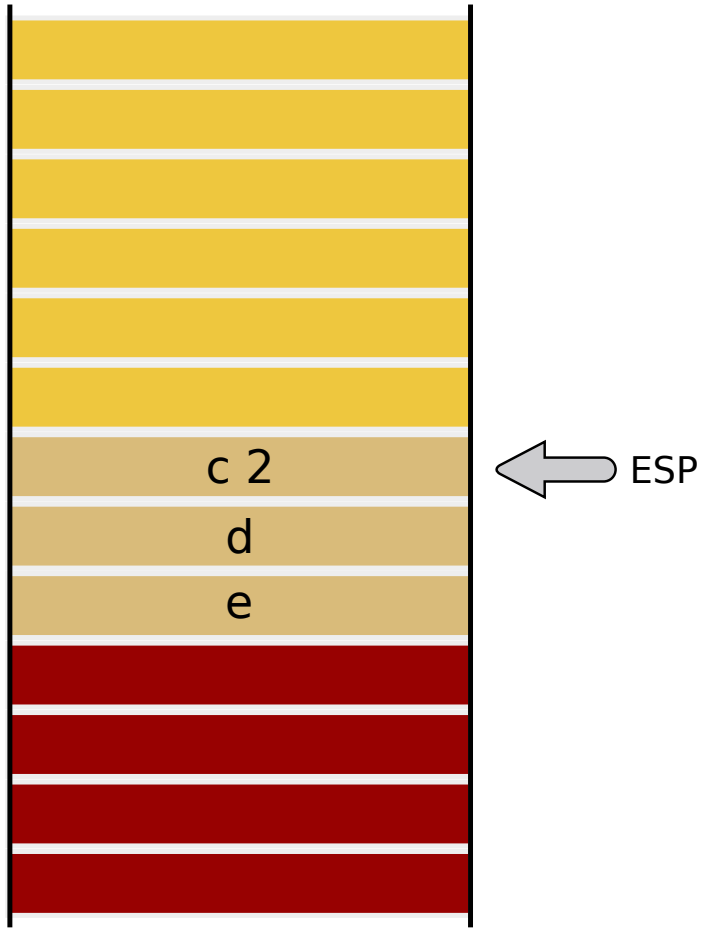


Ejemplo en 32 bits

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

llamado...

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

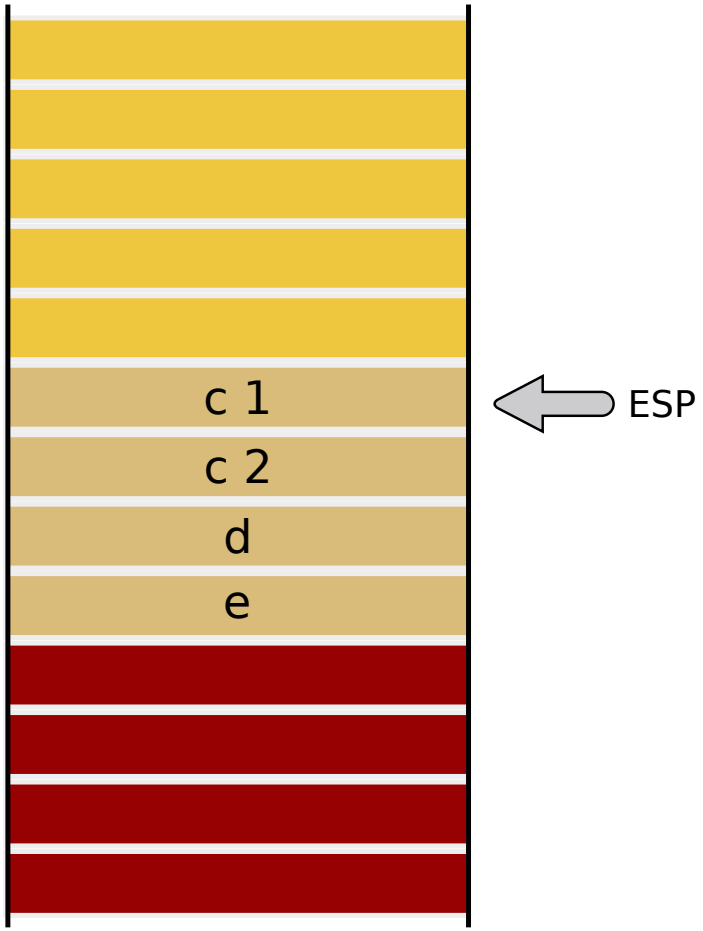


Ejemplo en 32 bits

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

llamado...

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

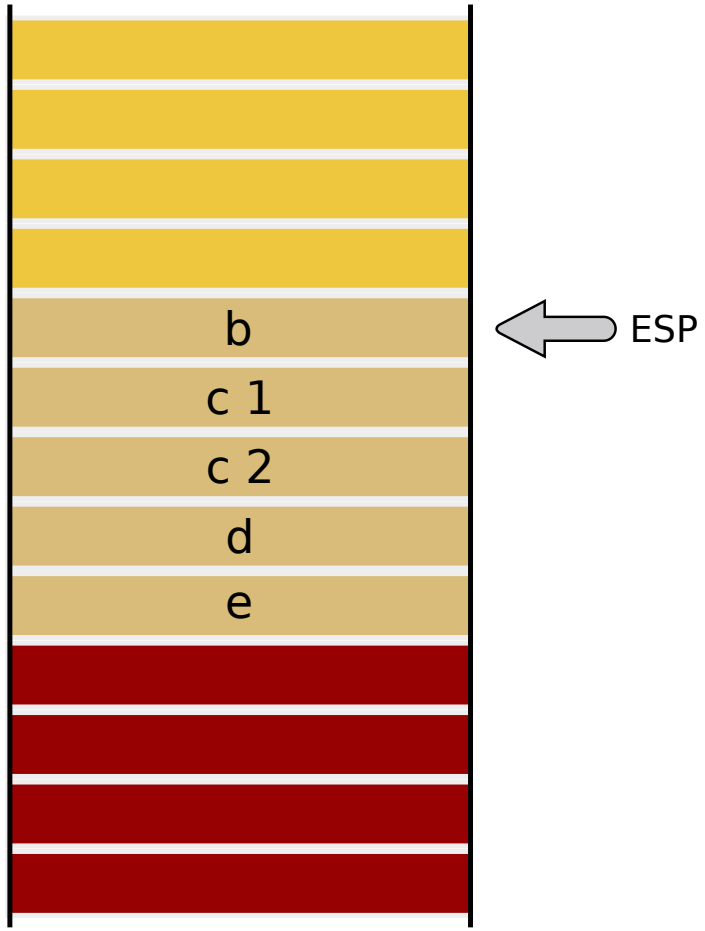


Ejemplo en 32 bits

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

llamado...

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

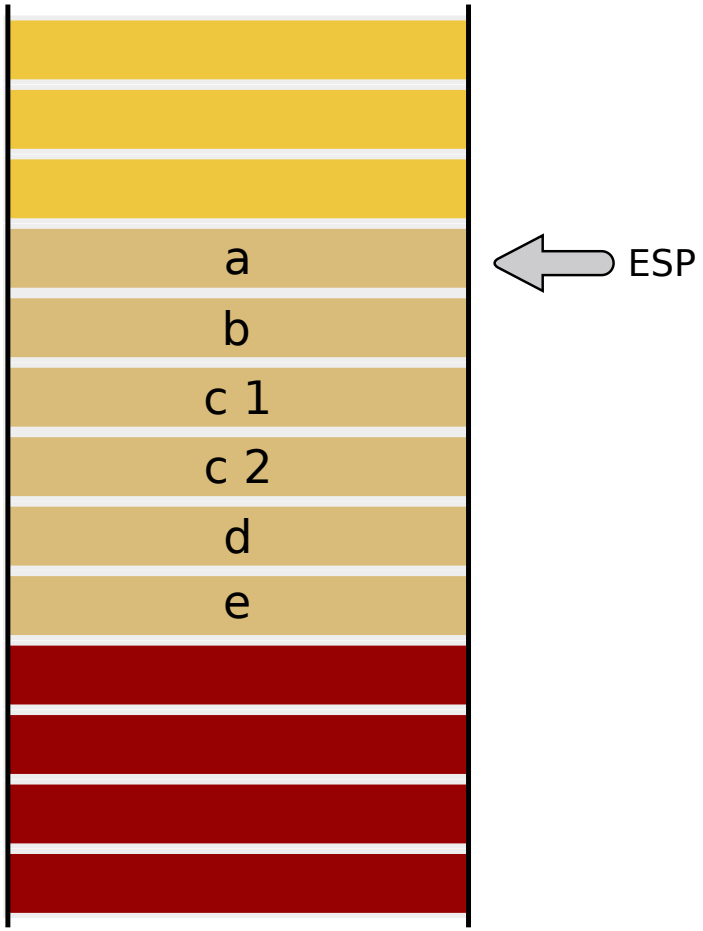


Ejemplo en 32 bits

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

llamado...

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

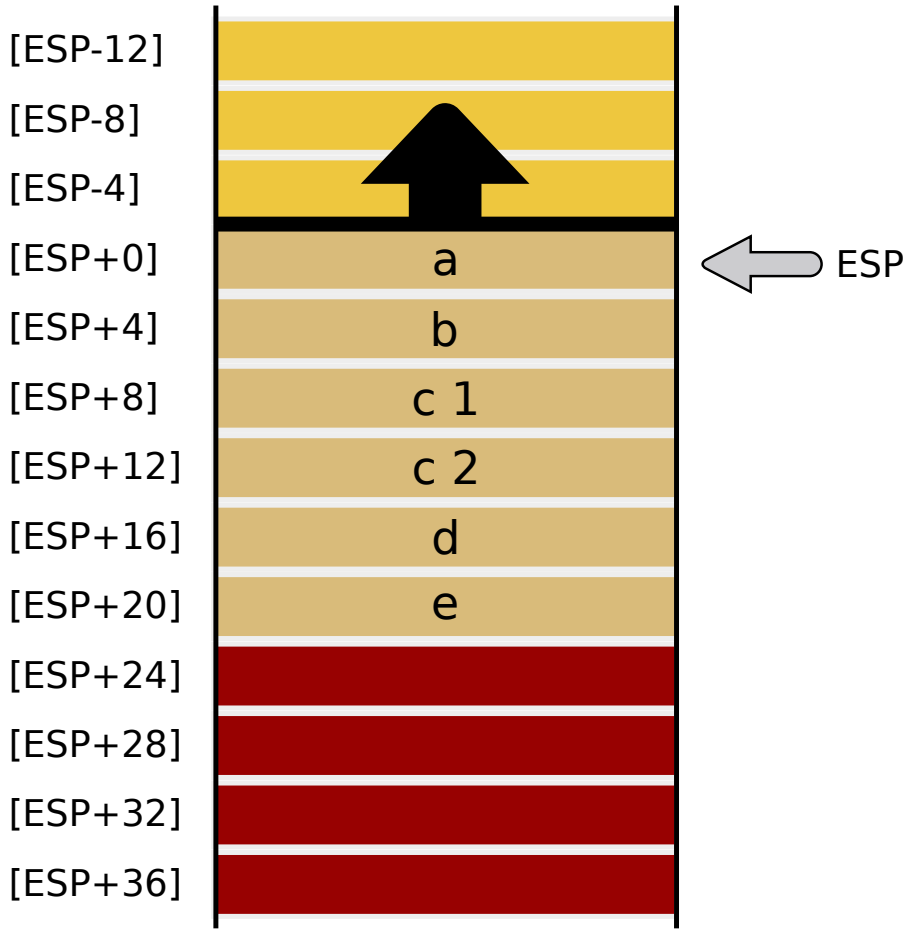


Ejemplo en 32 bits

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

llamado...

```
...
push e
push d
push c1
push c2
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)
```

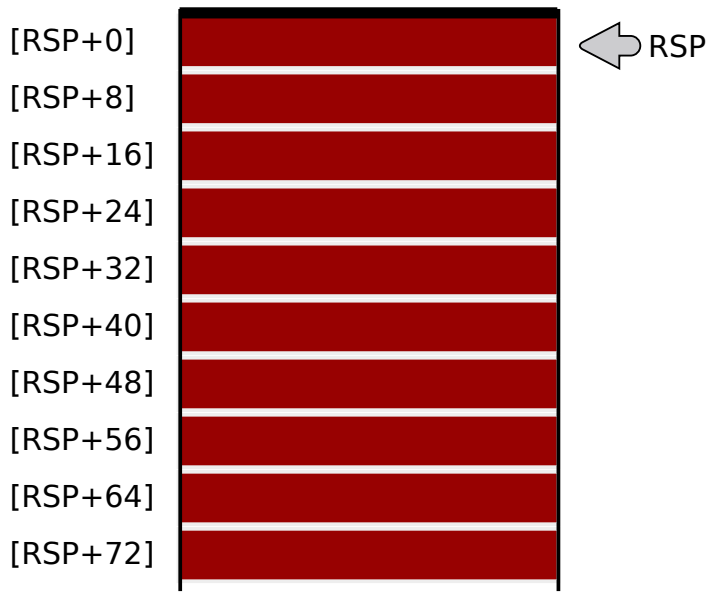
Enteros

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

Flotante

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

Pila



Ejemplo en 64 bits

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

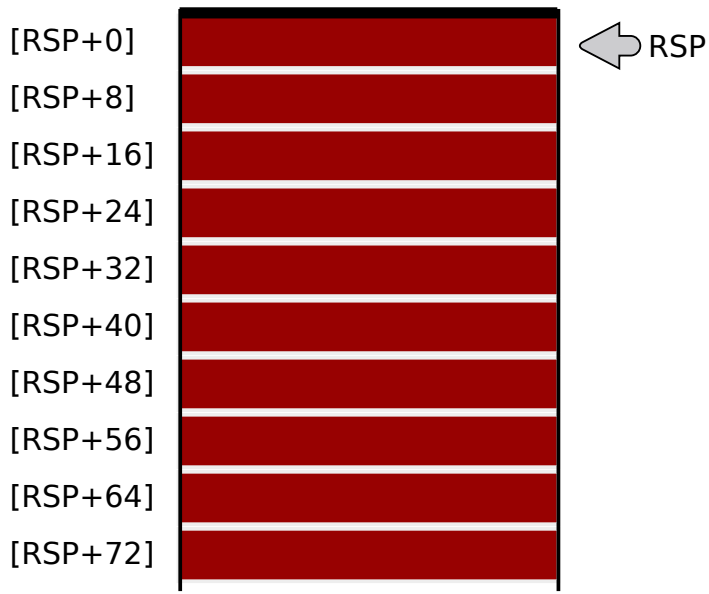
Enteros

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

Flotante

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

Pila



Ejemplo en 64 bits

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

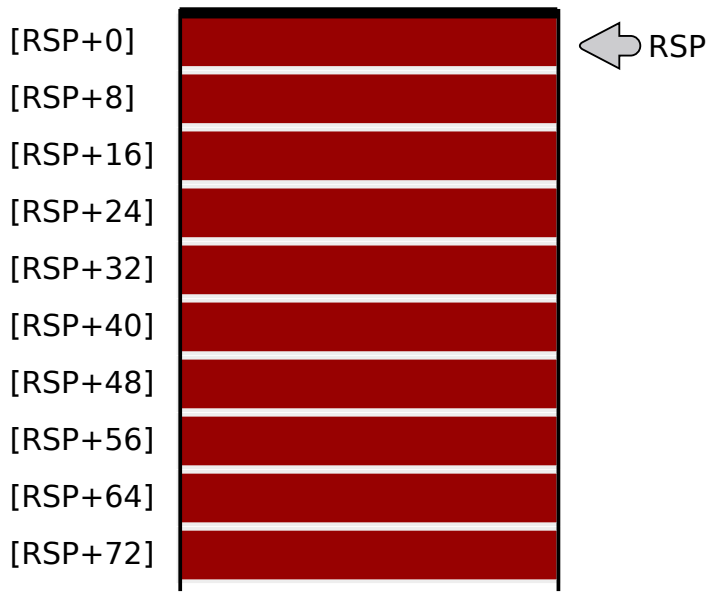
Enteros

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

Flotante

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

Pila



Ejemplo en 64 bits

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

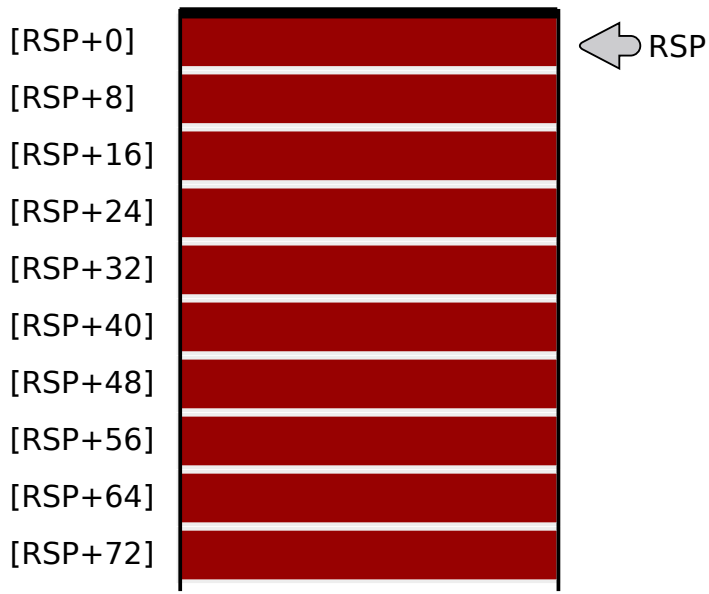
Enteros

```
RDI = a
RSI =
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)
```

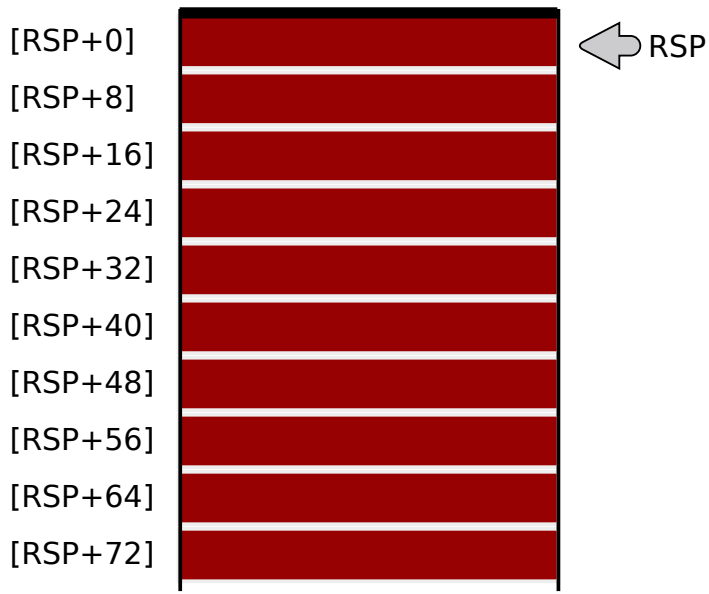
Enteros

```
RDI = a
RSI = d
RDX =
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)
```

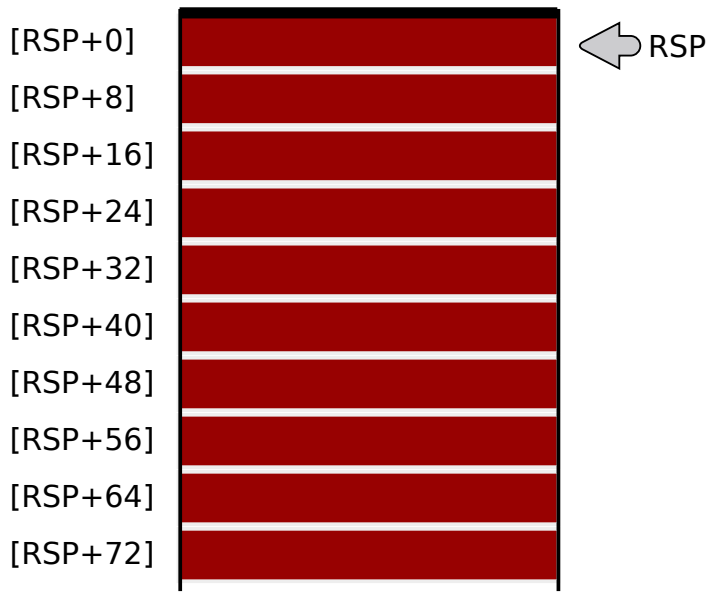
Enteros

```
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)
```

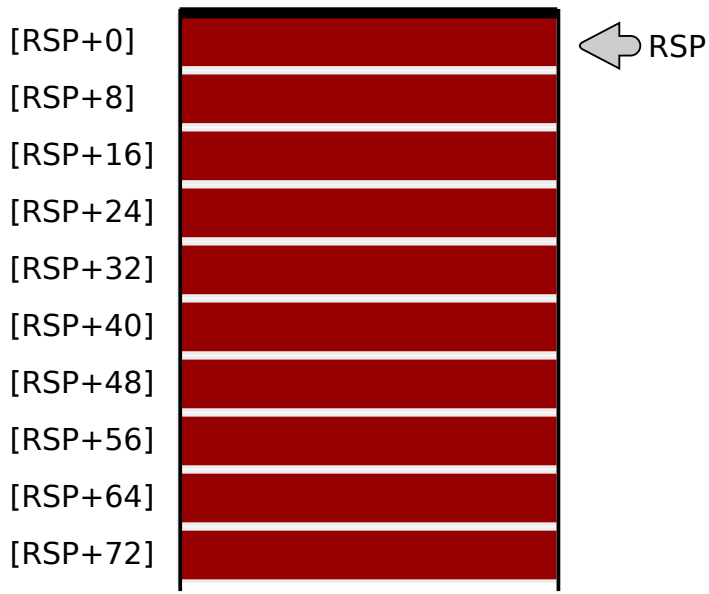
Enteros

```
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 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)
```

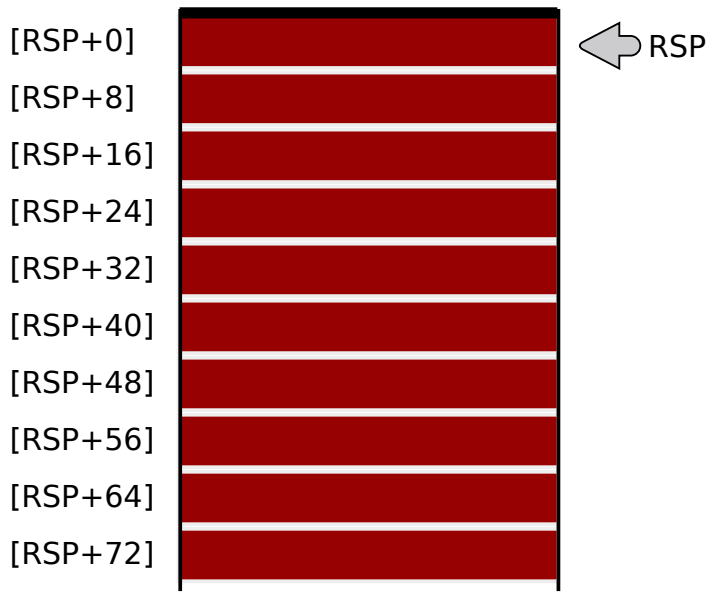
Enteros

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)
```

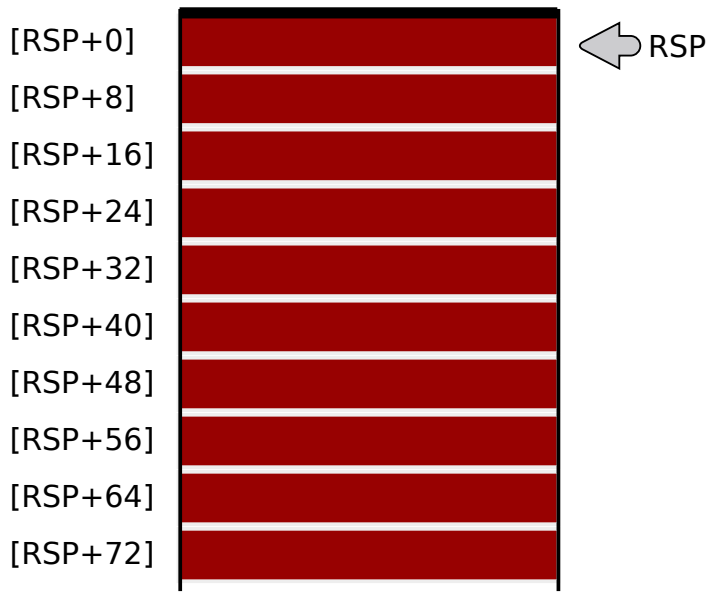
Enteros

```
RDI = a1
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)
```

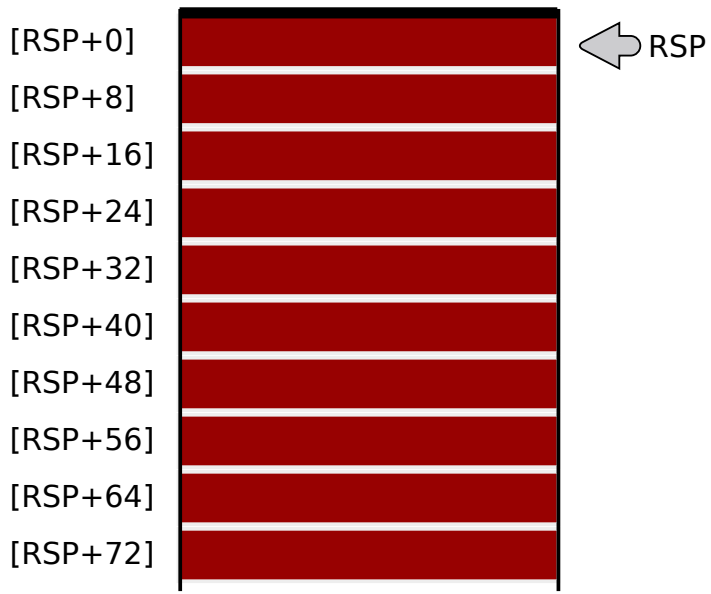
Enteros

```
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)
```

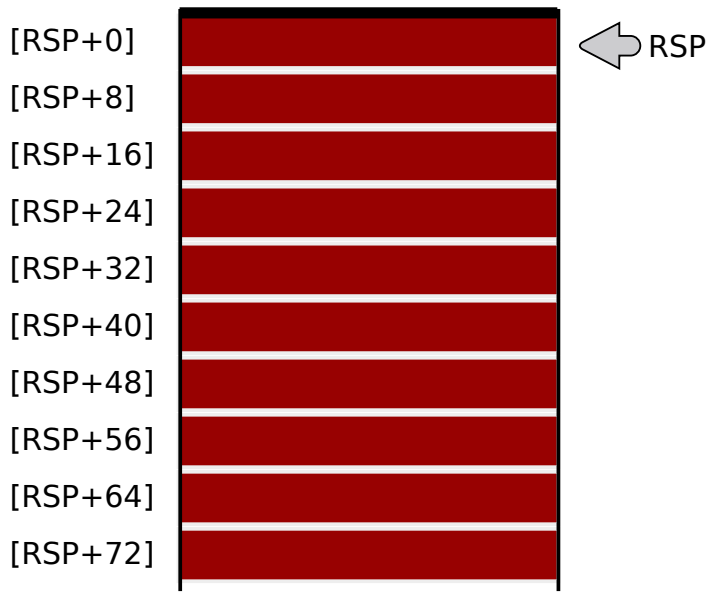
Enteros

```
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)
```

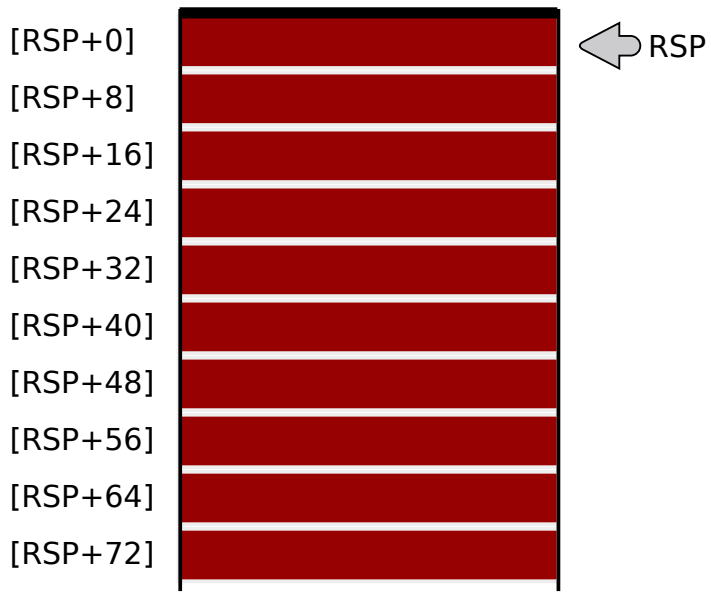
Enteros

```
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)
```

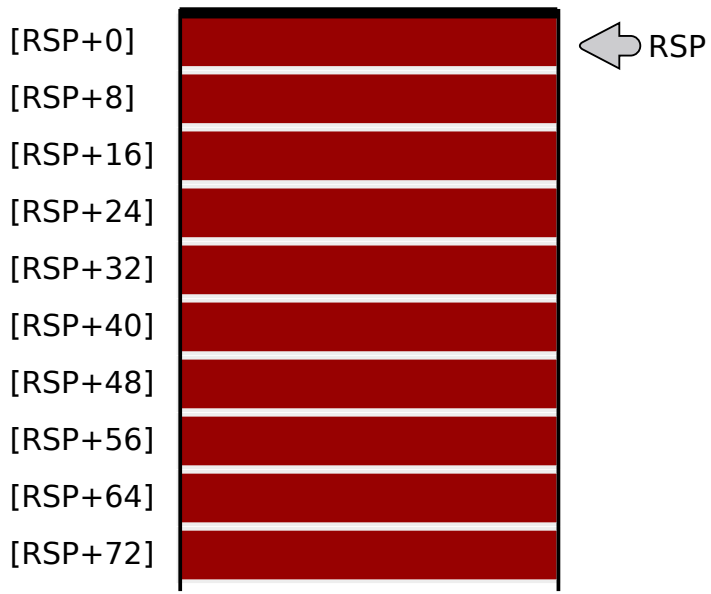
Enteros

```
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)
```

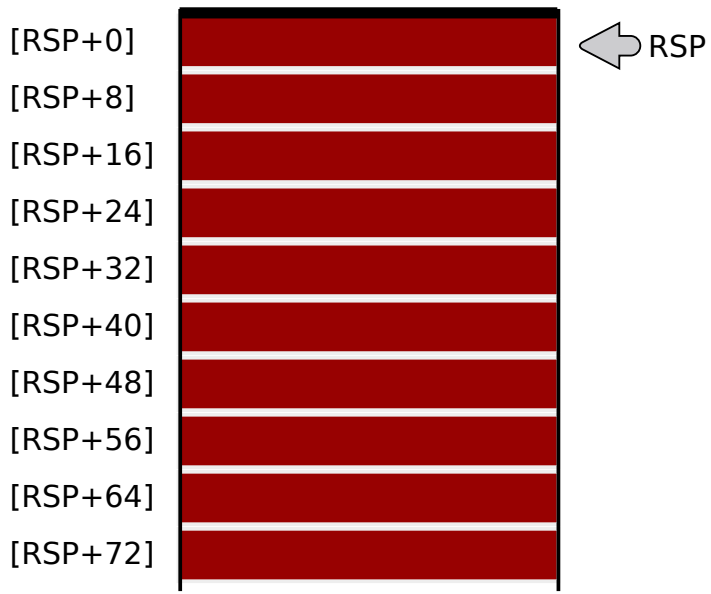
Enteros

```
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)
```

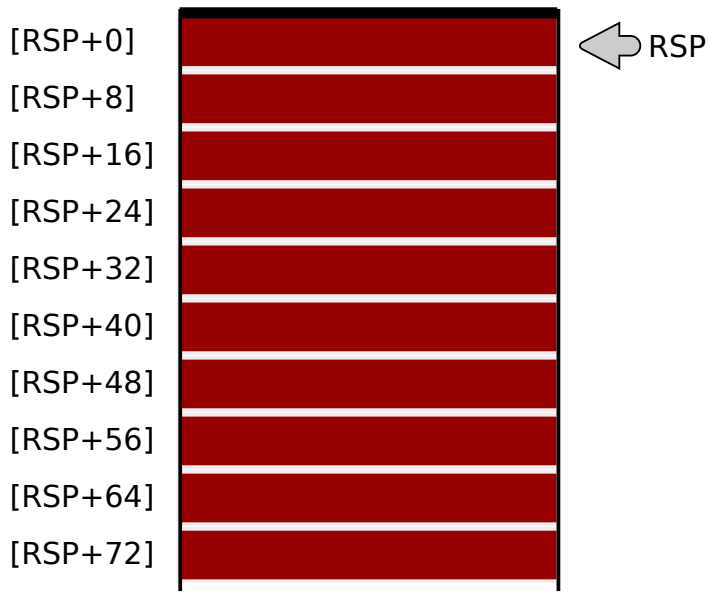
Enteros

```
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)
```

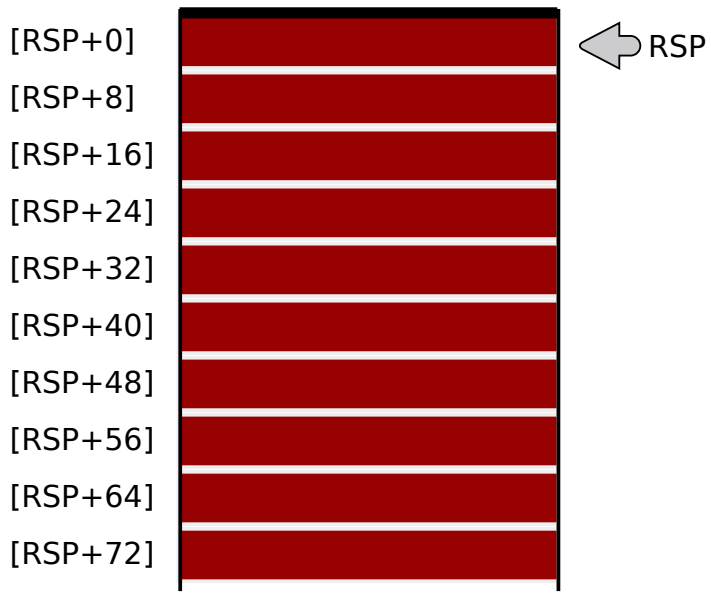
Enteros

```
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)
```

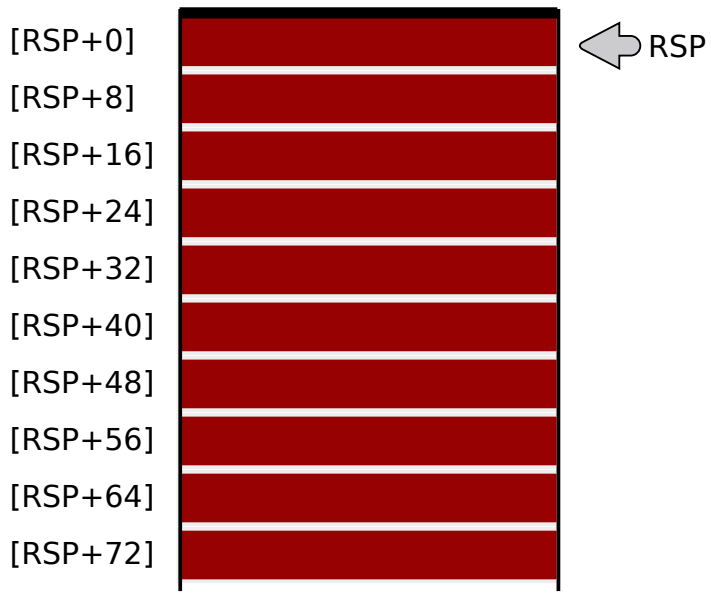
Enteros

```
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)
```

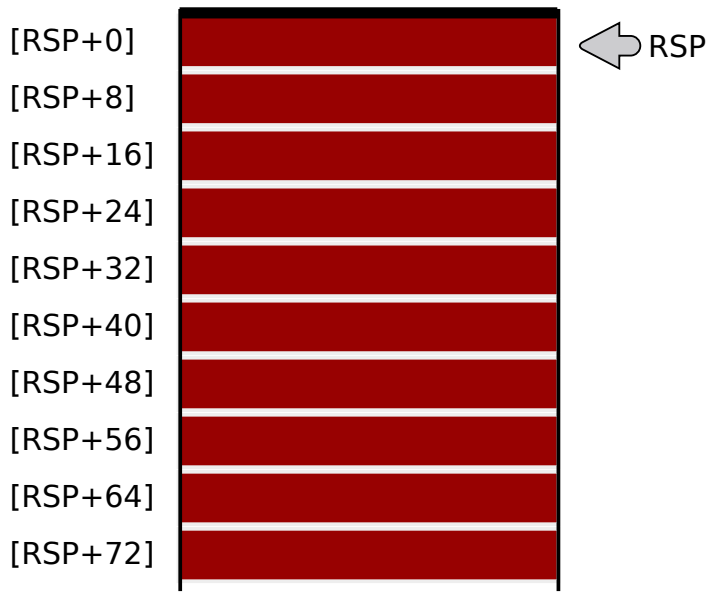
Enteros

```
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)
```

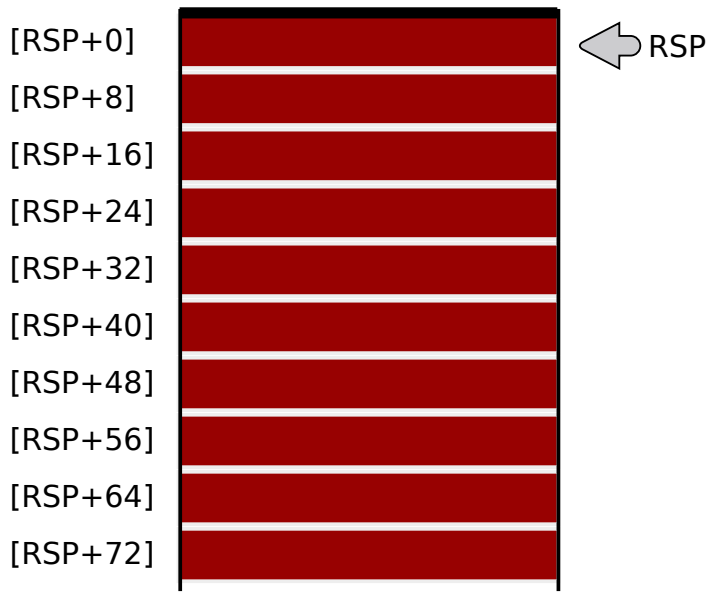
Enteros

```
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)
```

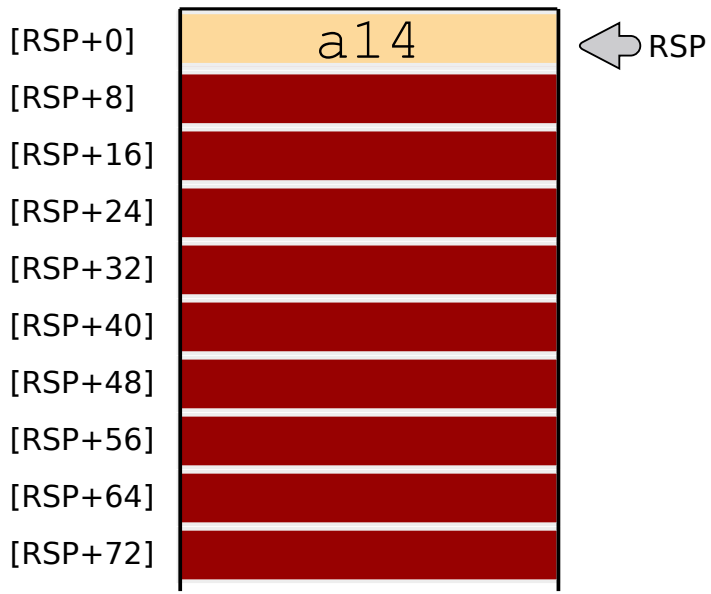
Enteros

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

Flotante

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

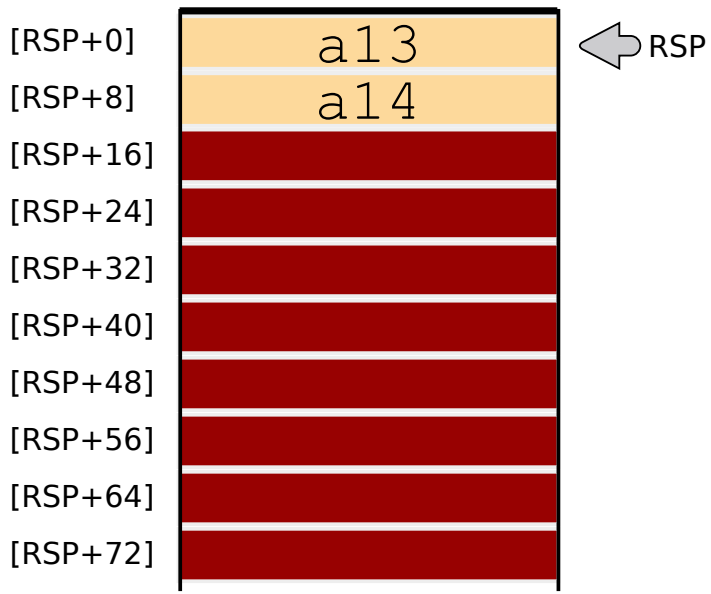
Enteros

```
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)
```

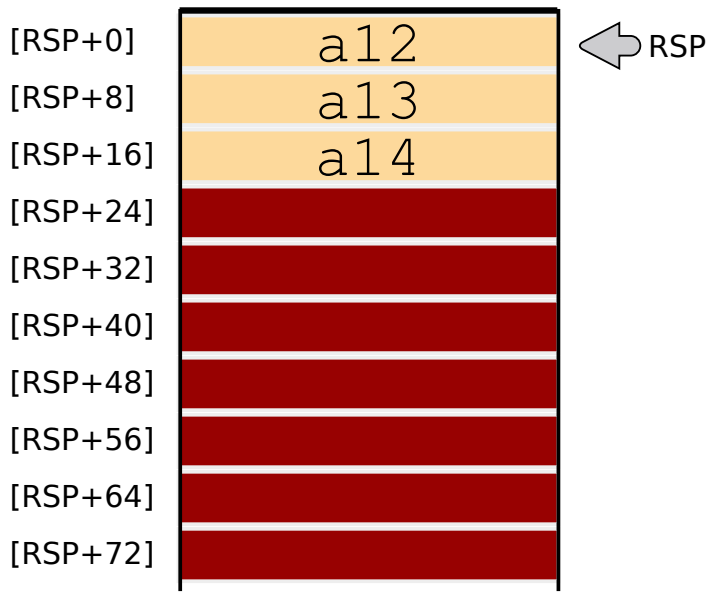
Enteros

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

Flotante

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

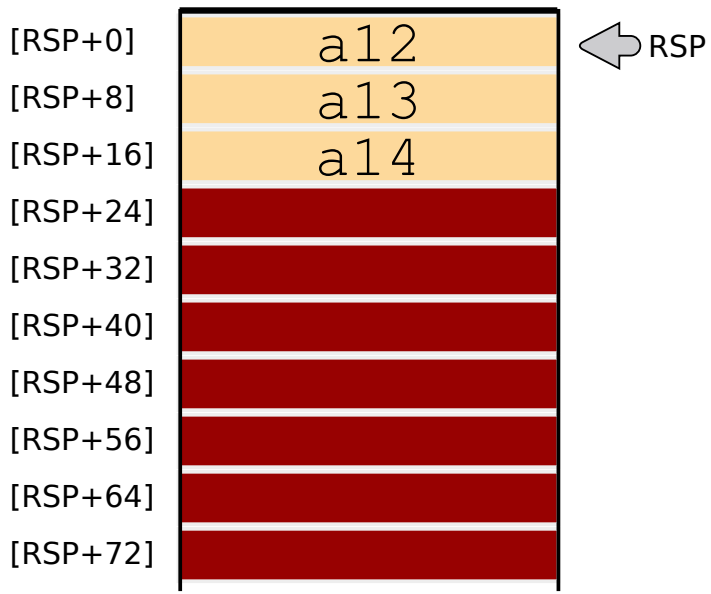
Enteros

```
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)
```

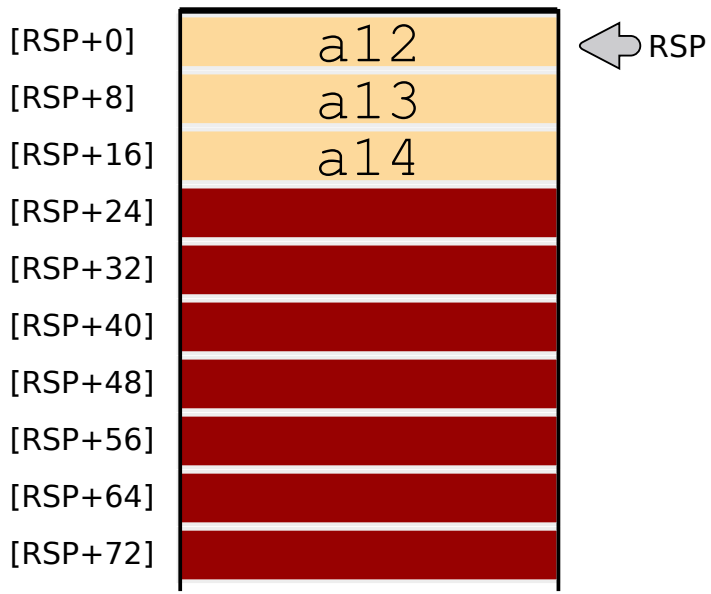
Enteros

```
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



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`

Ejercicios

1. Ensamblar y ejecutar el ejemplo de "`Hola Mundo`" en `ASM`.

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

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

4. 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 );
```

5. 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
```

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.

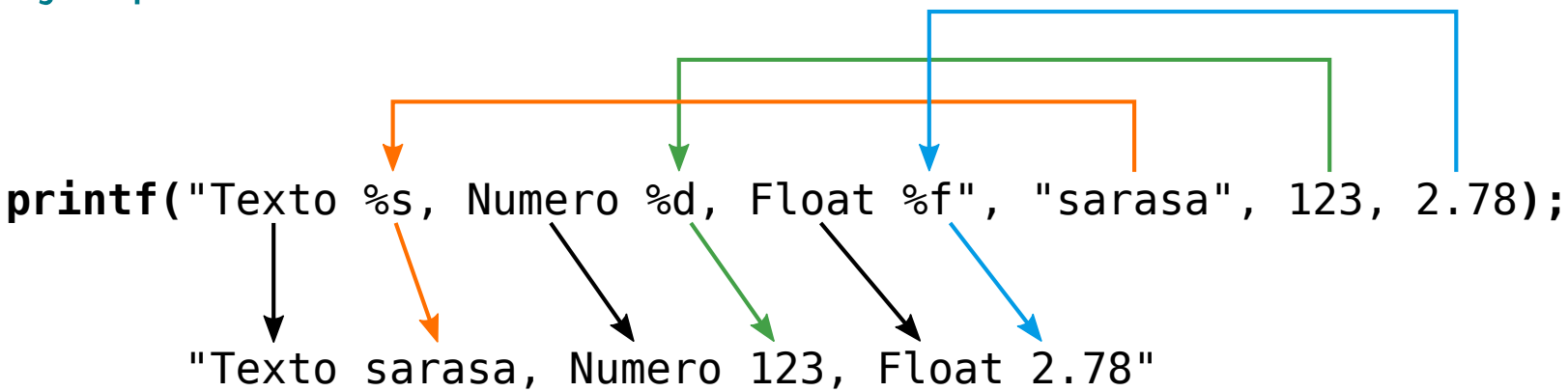
Funciones variádicas (de aridad variable)

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:



Desde ASM

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

¿Preguntas?

¡Gracias!