Práctica de x86 Assembly y ABI

Parte 2: ABI (System V)

Segundo Cuatrimestre 2024

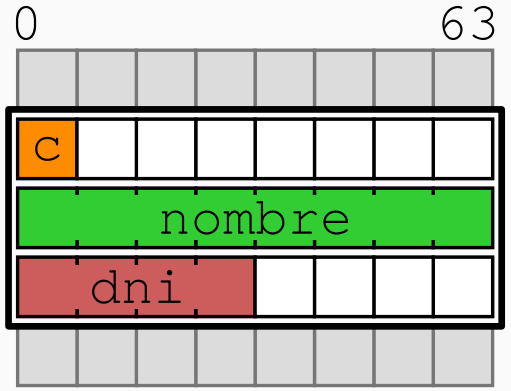
Arquitectura y Organización de Computadores
DC - UBA

Introducción a contratos

Introducción

Ya vimos un tipo de contrato:

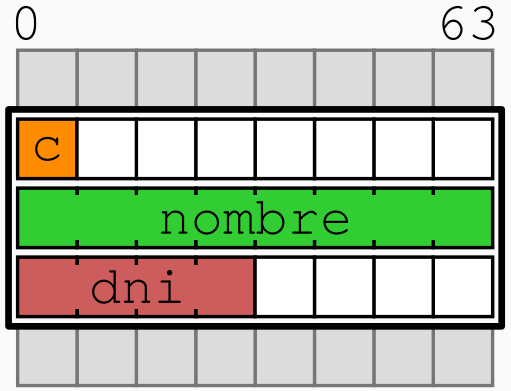
```
struct alumno{
        char c;
        char* nombre;
        int dni;
        };
```



Introducción

Ya vimos un tipo de contrato:

```
struct alumno{
        char c;
        char* nombre;
        int dni;
        };
```

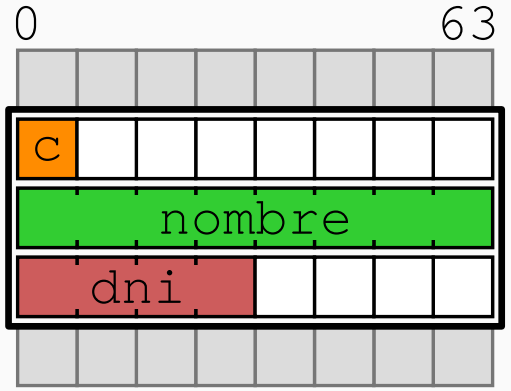


Es un **contrato de datos.**

Introducción

Ya vimos un tipo de contrato:

```
struct alumno{
        char c;
        char* nombre;
        int dni;
        };
```



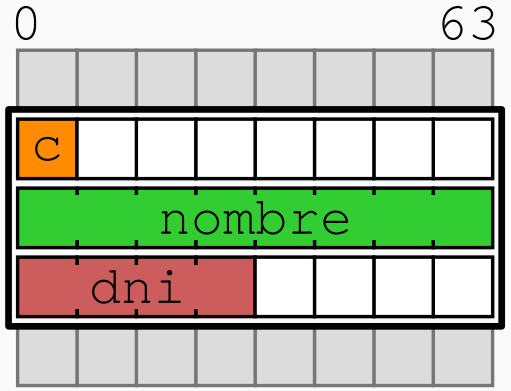
Es un **contrato de datos.**

Ahora veremos:

Introducción

Ya vimos un tipo de contrato:

```
struct alumno{
        char c;
        char* nombre;
        int dni;
        };
```



Es un **contrato de datos.**

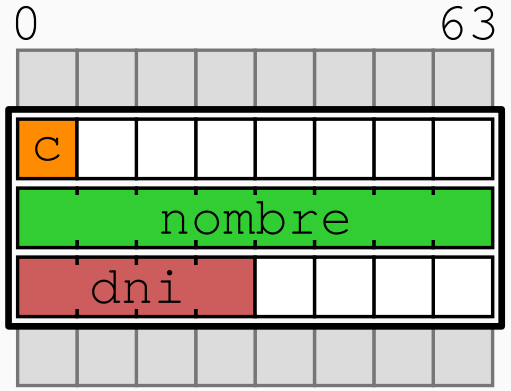
Ahora veremos:

- Contratos de funciones

Introducción

Ya vimos un tipo de contrato:

```
struct alumno{
        char c;
        char* nombre;
        int dni;
        };
```



Es un **contrato de datos.**

Ahora veremos:

- Contratos de funciones
- Convenciones de registros y pila

Contratos de funciones

¿Qué es una declaración de función en C?

```
int32_t product(int32_t *arr, uint32_t length);
```

Contratos de funciones

¿Qué es una declaración de función en C?

```c
int32_t product(int32_t *arr, uint32_t length);
```

- Existe una función llamada `product`.

Contratos de funciones

¿Qué es una declaración de función en C?

```c
int32_t product(int32_t *arr, uint32_t length);
```

- Existe una función llamada `product`.
- Quien llame a `product` va a obtener un valor de tipo `int32_t`.

Contratos de funciones

¿Qué es una declaración de función en C?

```
int32_t product(int32_t *arr, uint32_t length);
```

- Existe una función llamada `product`.
- Quien llame a `product` va a obtener un valor de tipo `int32_t`.
- Para llamar a `product` hay que proporcionar dos valores: uno de tipo `int32_t*` y otro de tipo `uin32_t`.

Contratos de funciones

¿Qué es una declaración de función en C?

```
int32_t product(int32_t *arr, uint32_t length);
```

- Todo uso o implementación de la función **deben coincidir en tipo devuelto, cantidad y tipo de parámetros**.

Contratos de funciones

¿Qué es una declaración de función en C?

```c
int32_t product(int32_t *arr, uint32_t length);
```

- Todo uso o implementación de la función **deben coincidir en tipo devuelto, cantidad y tipo de parámetros**.
- Este contrato es respetado automáticamente por el compilador de C (`gcc`, `clang`, etc).

Contratos de funciones

¿Qué es una declaración de función en C?

```
int32_t product(int32_t *arr, uint32_t length);
```

- Todo uso o implementación de la función **deben coincidir en tipo devuelto, cantidad y tipo de parámetros**.
- Este contrato es respetado automáticamente por el compilador de C (`gcc`, `clang`, etc).
- ¿Qué sucede cuando queremos llamar a una función de C desde ASM o a una función de ASM desde C?

Contratos de funciones

¿Qué sucede cuando queremos llamar a una función de `C` desde `ASM` o a una función de `ASM` desde `C`?

Contratos de funciones

¿Qué sucede cuando queremos llamar a una función de `C` desde `ASM` o a una función de `ASM` desde `C`?

Respuesta: Depende de la **plataforma**.
Es decir, vamos a tener que definir el alcance de nuestro contrato en términos de la plataforma particular.

Contratos de funciones

¿Qué sucede cuando queremos llamar a una función de `C` desde `ASM` o a una función de `ASM` desde `C`?

Respuesta: Depende de la **plataforma**.
Es decir, vamos a tener que definir el alcance de nuestro contrato en términos de la plataforma particular.

Corolario: Los contratos de función en un lenguaje de alto nivel (i.e. API) se pueden definir independientemente de la plataforma.

Contratos de funciones

¿Entonces cualquier lenguaje que genere **código objeto y respete el contrato de función** puede interactuar con funciones ubicadas en bibliotecas binarias (código objeto) que adhieran al contrato?

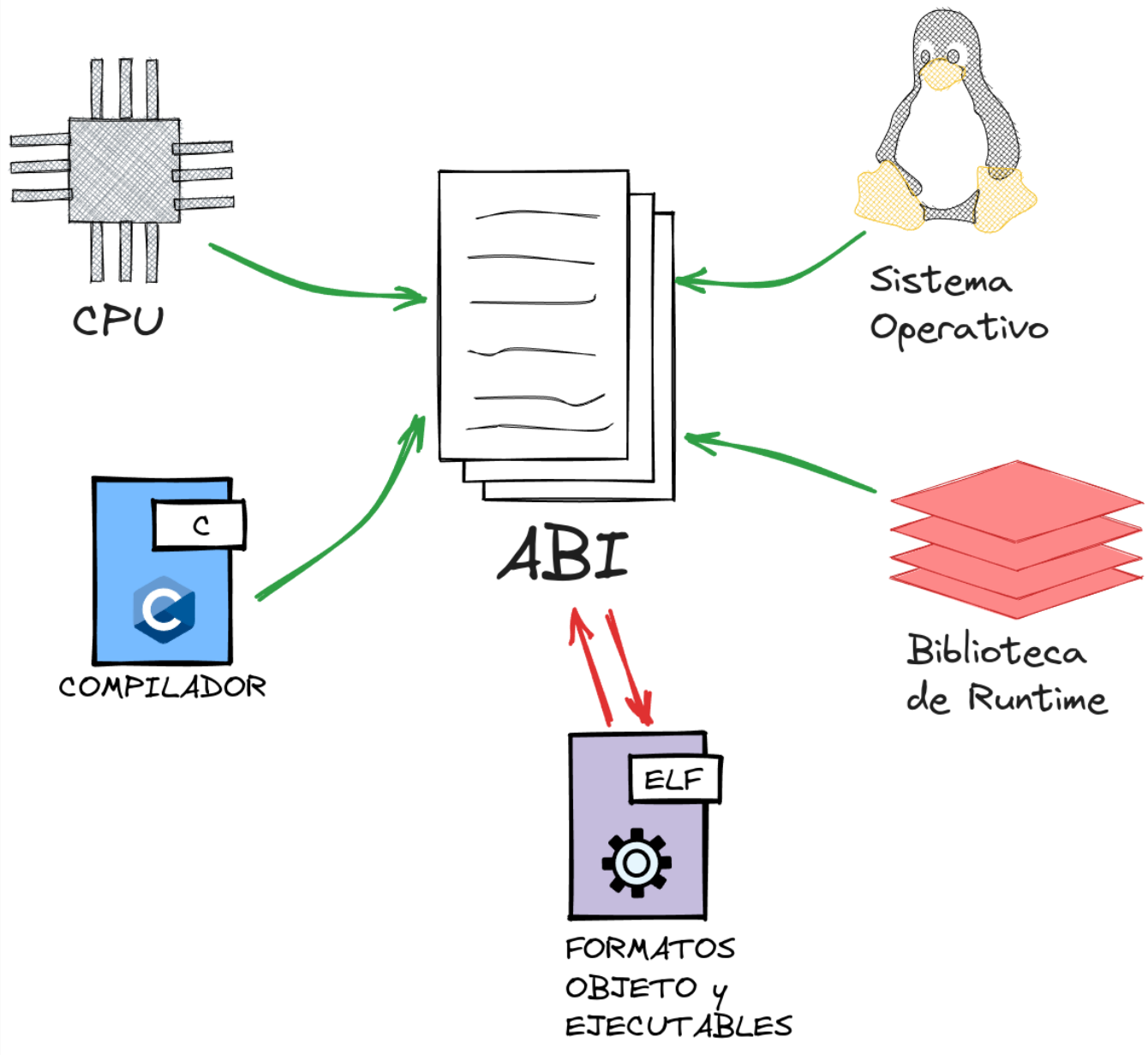
Contratos de funciones

¿Entonces cualquier lenguaje que genere **código objeto y respete el contrato de función** puede interactuar con funciones ubicadas en bibliotecas binarias (código objeto) que adhieran al contrato?

Respuesta: Si, para eso vamos a tener que hablar de la **ABI (Application Binary Interface)**

Interfaz binaria de aplicación (ABI)

Interfaz binaria de aplicación



Interfaz binaria de aplicación

Una **ABI** va a definir contratos sobre otros elementos que no nos interesan en este momento, como por ejemplo:

Interfaz binaria de aplicación

Una **ABI** va a definir contratos sobre otros elementos que no nos interesan en este momento, como por ejemplo:

- Formato de archivos objeto y ejecutables

Interfaz binaria de aplicación

Una **ABI** va a definir contratos sobre otros elementos que no nos interesan en este momento, como por ejemplo:

- Formato de archivos objeto y ejecutables
- Uso de bibliotecas compartidas

Interfaz binaria de aplicación

Una **ABI** va a definir contratos sobre otros elementos que no nos interesan en este momento, como por ejemplo:

- Formato de archivos objeto y ejecutables
- Uso de bibliotecas compartidas
- Parámetros pasados al proceso

Interfaz binaria de aplicación

Una **ABI** va a definir contratos sobre otros elementos que no nos interesan en este momento, como por ejemplo:

- Formato de archivos objeto y ejecutables
- Uso de bibliotecas compartidas
- Parámetros pasados al proceso
- Ubicación de tablas globales del sistema

Interfaz binaria de aplicación

Si bien son importantes no vamos a hablar de esos contratos en esta materia. Los que sí nos importan son:

Interfaz binaria de aplicación

Si bien son importantes no vamos a hablar de esos contratos en esta materia. Los que sí nos importan son:

- El set de instrucciones

Interfaz binaria de aplicación

Si bien son importantes no vamos a hablar de esos contratos en esta materia. Los que sí nos importan son:

- El set de instrucciones
- Tamaño y alineación de los tipos de datos primitivos

Interfaz binaria de aplicación

Si bien son importantes no vamos a hablar de esos contratos en esta materia. Los que sí nos importan son:

- El set de instrucciones
- Tamaño y alineación de los tipos de datos primitivos
- La forma de enviar y recibir información usando funciones del sistema (Convención de llamada para **System Calls**)

Interfaz binaria de aplicación

Si bien son importantes no vamos a hablar de esos contratos en esta materia. Los que sí nos importan son:

- El set de instrucciones
- Tamaño y alineación de los tipos de datos primitivos
- La forma de enviar y recibir información usando funciones del sistema (Convención de llamada para **System Calls**)
- La forma de enviar y recibir información usando funciones de usuarix (**Convención de llamada**)

ABI: System V

La **ABI** que utilizaremos define dos convenciones:

ABI: System V

La **ABI** que utilizaremos define dos convenciones:

- En `x86-64/Linux` (64bits) se denomina `System V AMD64 ABI`
- En `x86/Linux` (32bits) se conoce como `System V i386 ABI`

ABI: System V

La **ABI** que utilizaremos define dos convenciones:

- En `x86-64/Linux` (64bits) se denomina `System V AMD64 ABI`
- En `x86/Linux` (32bits) se conoce como `System V i386 ABI`

Vamos a usar el primero ahora haciendo programación de aplicaciones y el segundo cuando hagamos programación de sistemas.

Convención de llamadas

Llamadas a funciones

Una función:

En C:

```
1  int32_t suma(int32_t x,
2               int32_t y){
3          return x+y;
4  }
5
6
```

```
1  ...
2  a = suma(c+d);
3  ...
4
5
6
```

En ASM:

```
1  suma: push rbp
2        mov rbp, rsp
3        add esi, edi
4        mov eax, esi
5        pop rbp
6        ret
```

```
1  ...
2  mov edi, [rbp-OFFSET_X]
3  mov esi, [rbp-OFFSET_Y]
4  call suma
5  mov [rbp-OFFSET_Z], eax
6  ...
```

Llamadas a funciones

Una función:

En C:

```c
int32_t suma(int32_t x,
             int32_t y){
        return x+y;
}
```

En ASM:

```asm
suma: push rbp
      mov rbp, rsp
      add esi, edi
      mov eax, esi
      pop rbp
      ret
```

```c
...
a = suma(c+d);
...
```

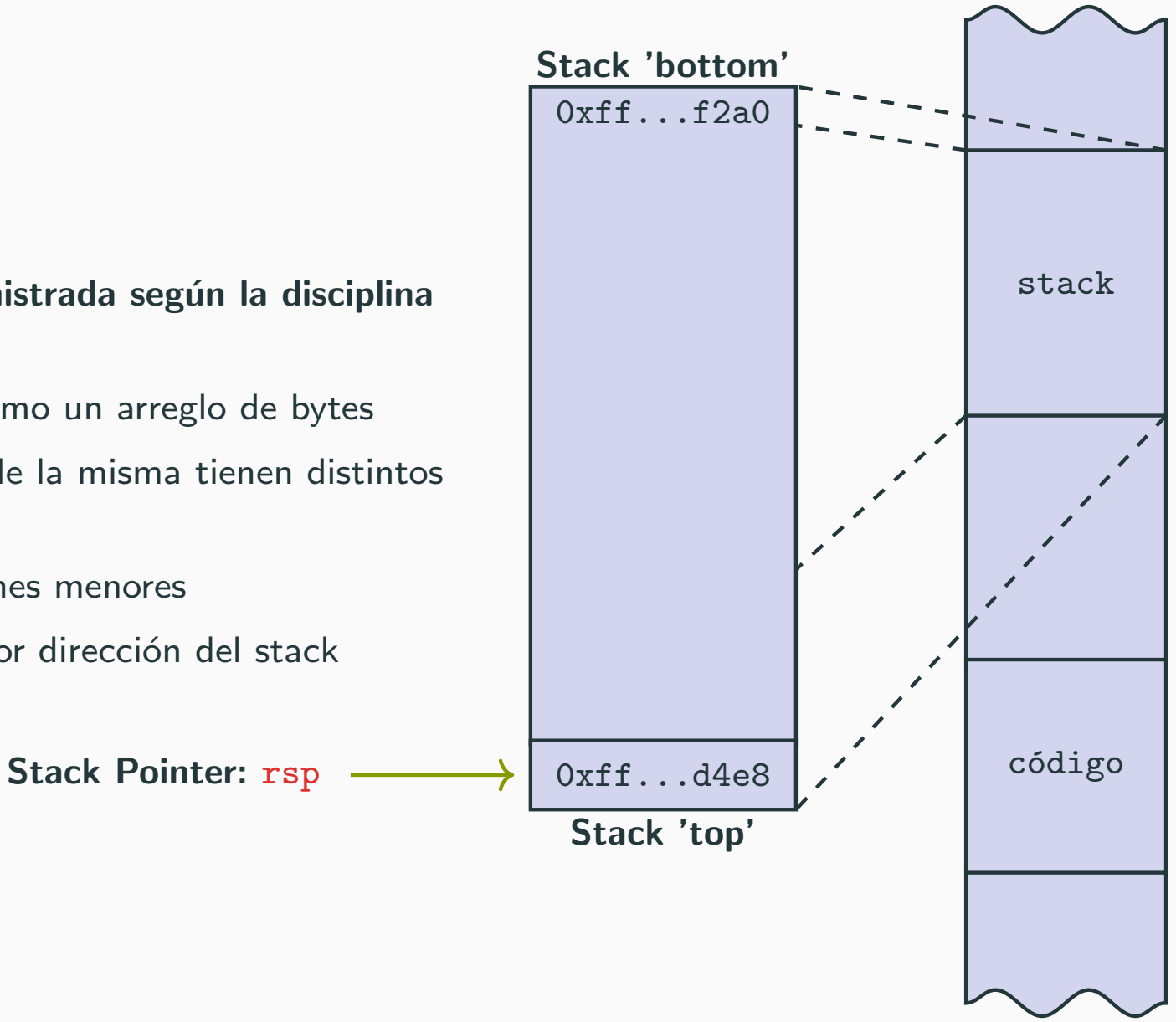
```asm
...
mov edi, [rbp-OFFSET_X]
mov esi, [rbp-OFFSET_Y]
call suma
mov [rbp-OFFSET_Z], eax
...
```

Claramente necesitamos entender **la pila o stack**.

Pila (Stack) x86-64

Región de memoria administrada según la disciplina del stack

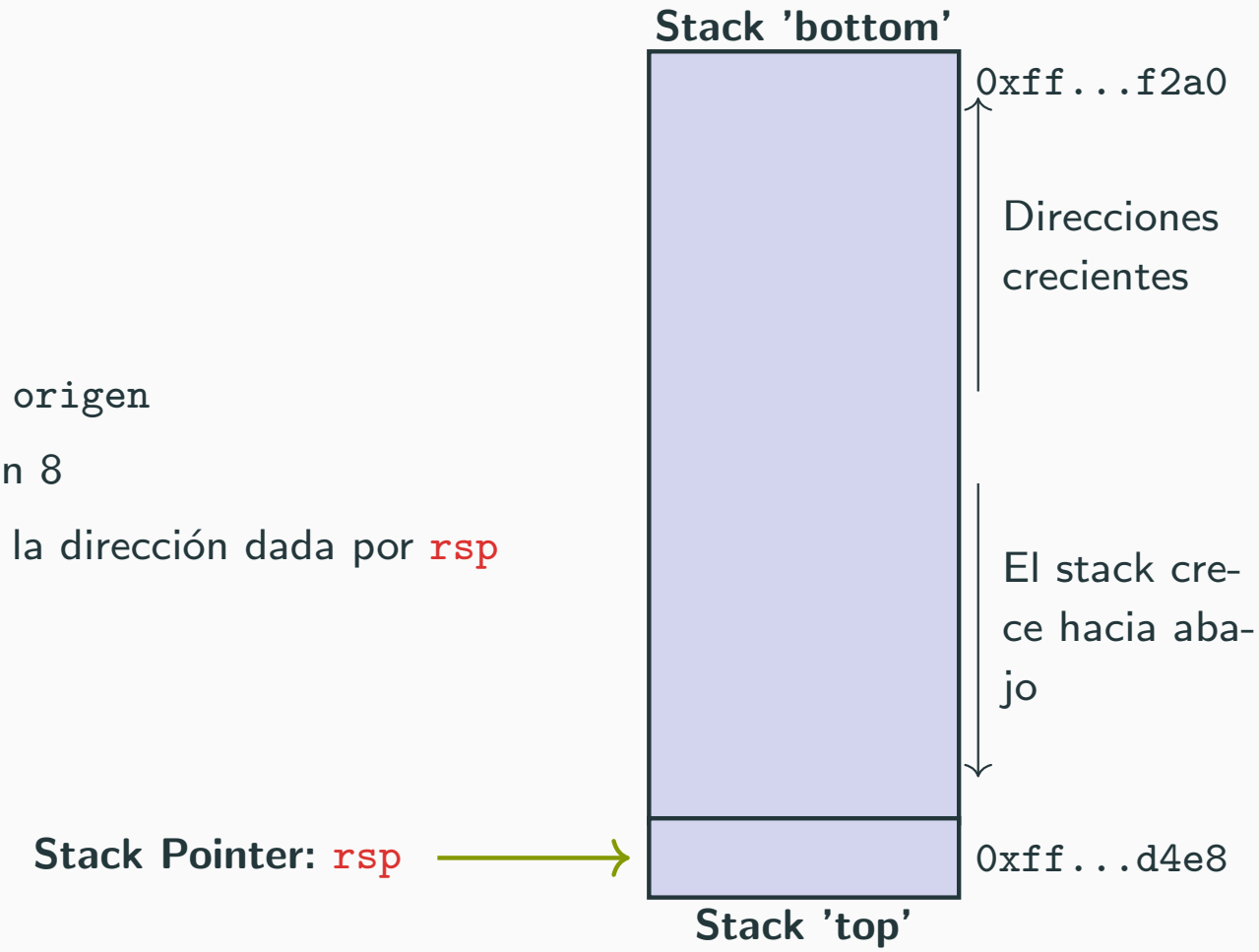
- La memoria se ve como un arreglo de bytes
- Diferentes regiones de la misma tienen distintos propósitos
- Crece hacia direcciones menores
- `rsp` contiene la menor dirección del stack



Stack x86-64: push

`push origen`

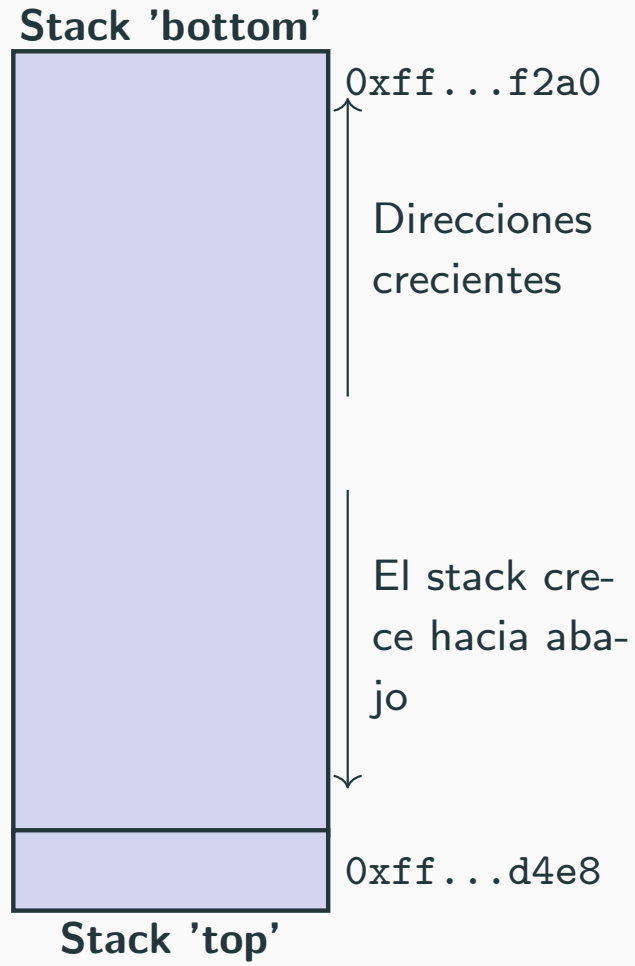
- obtener el dato de `origen`
- decrementar `rsp` en 8
- guardar el dato en la dirección dada por `rsp`



Stack x86-64: push

`push origen`

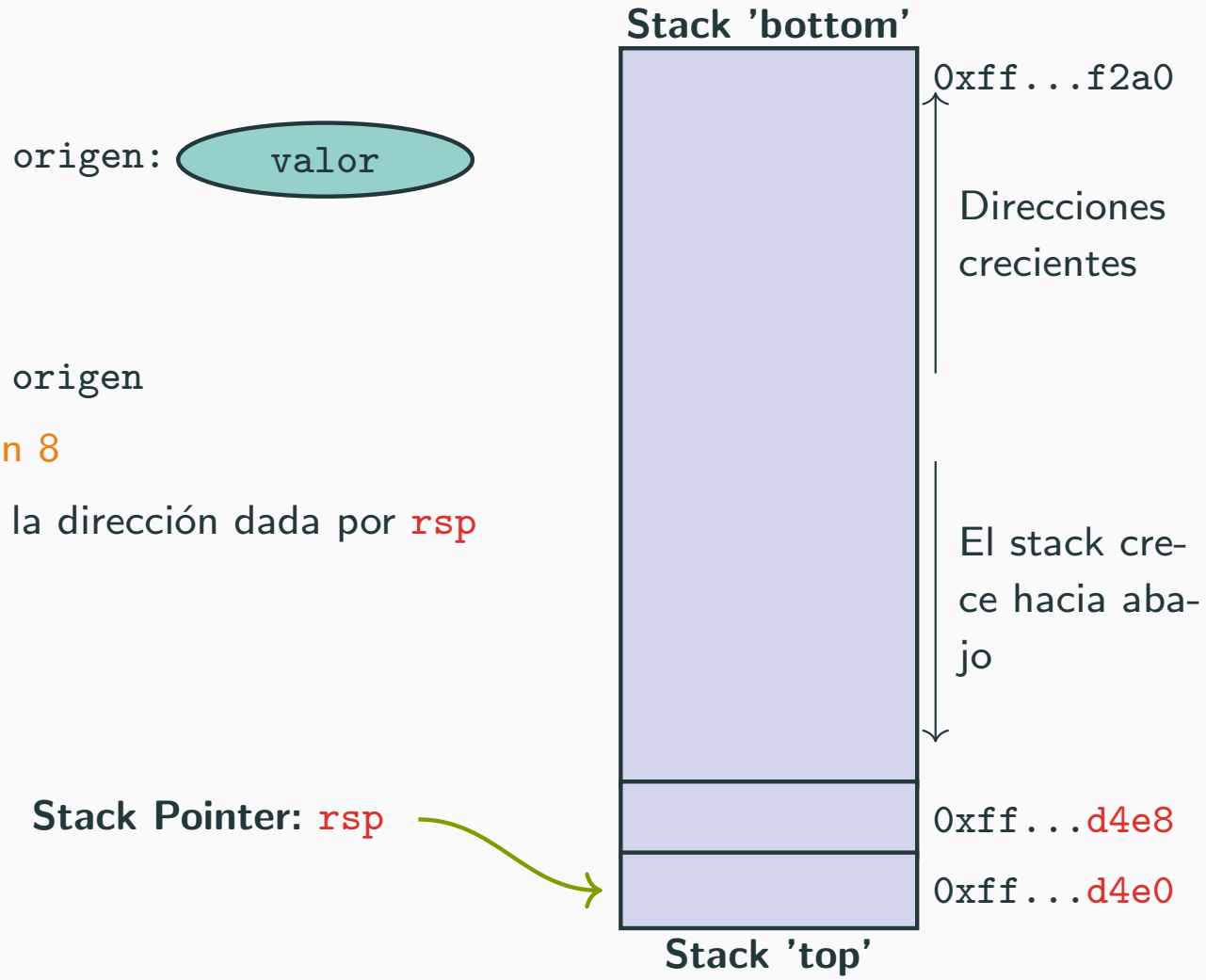
- obtener el dato de `origen`
- decrementar `rsp` en 8
- guardar el dato en la dirección dada por `rsp`



Stack x86-64: push

`push origen`

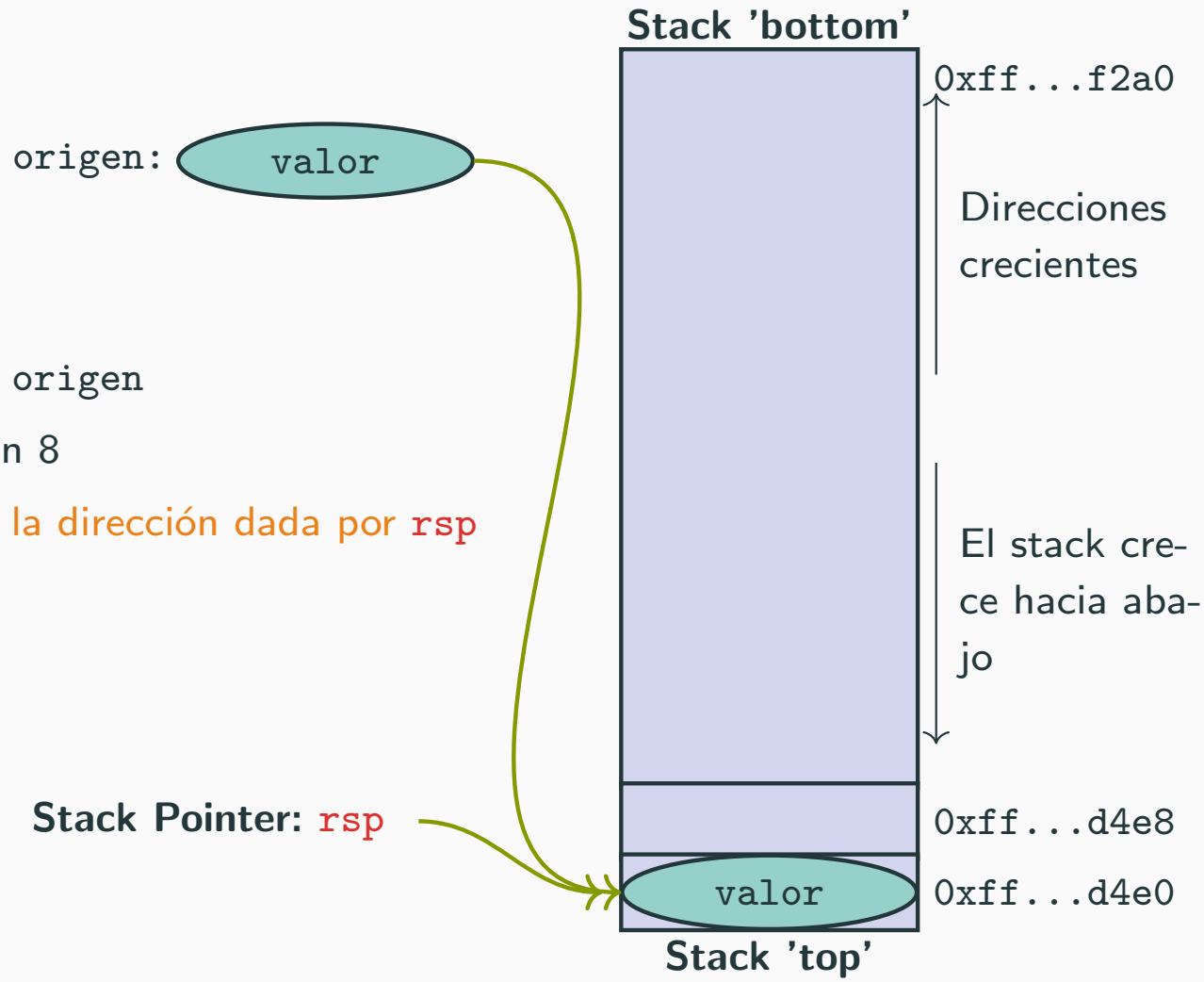
- obtener el dato de `origen`
- decrementar `rsp` en 8
- guardar el dato en la dirección dada por `rsp`



Stack x86-64: push

`push origen`

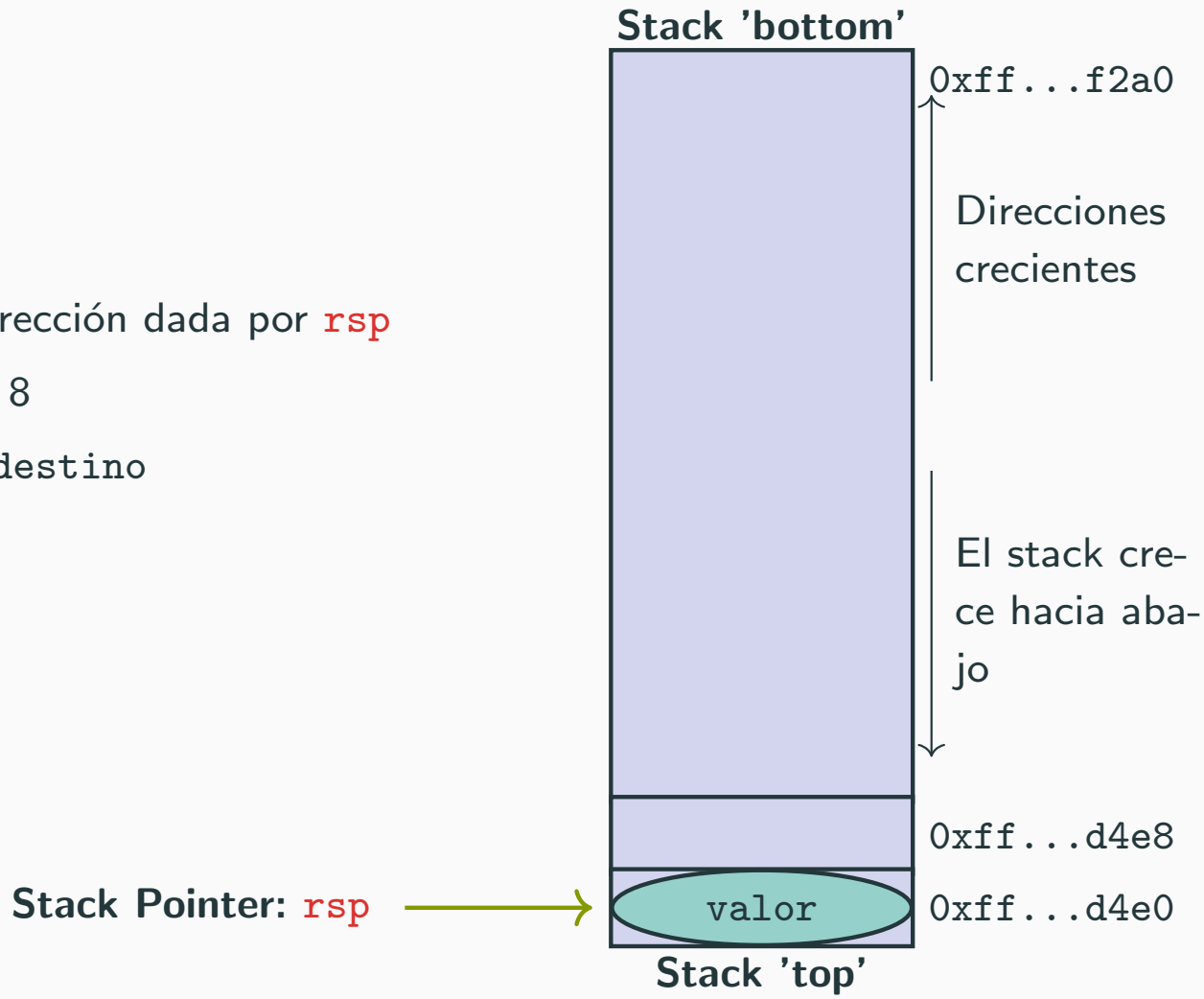
- obtener el dato de `origen`
- decrementar `rsp` en 8
- guardar el dato en la dirección dada por `rsp`



Stack x86-64: pop

`pop destino`

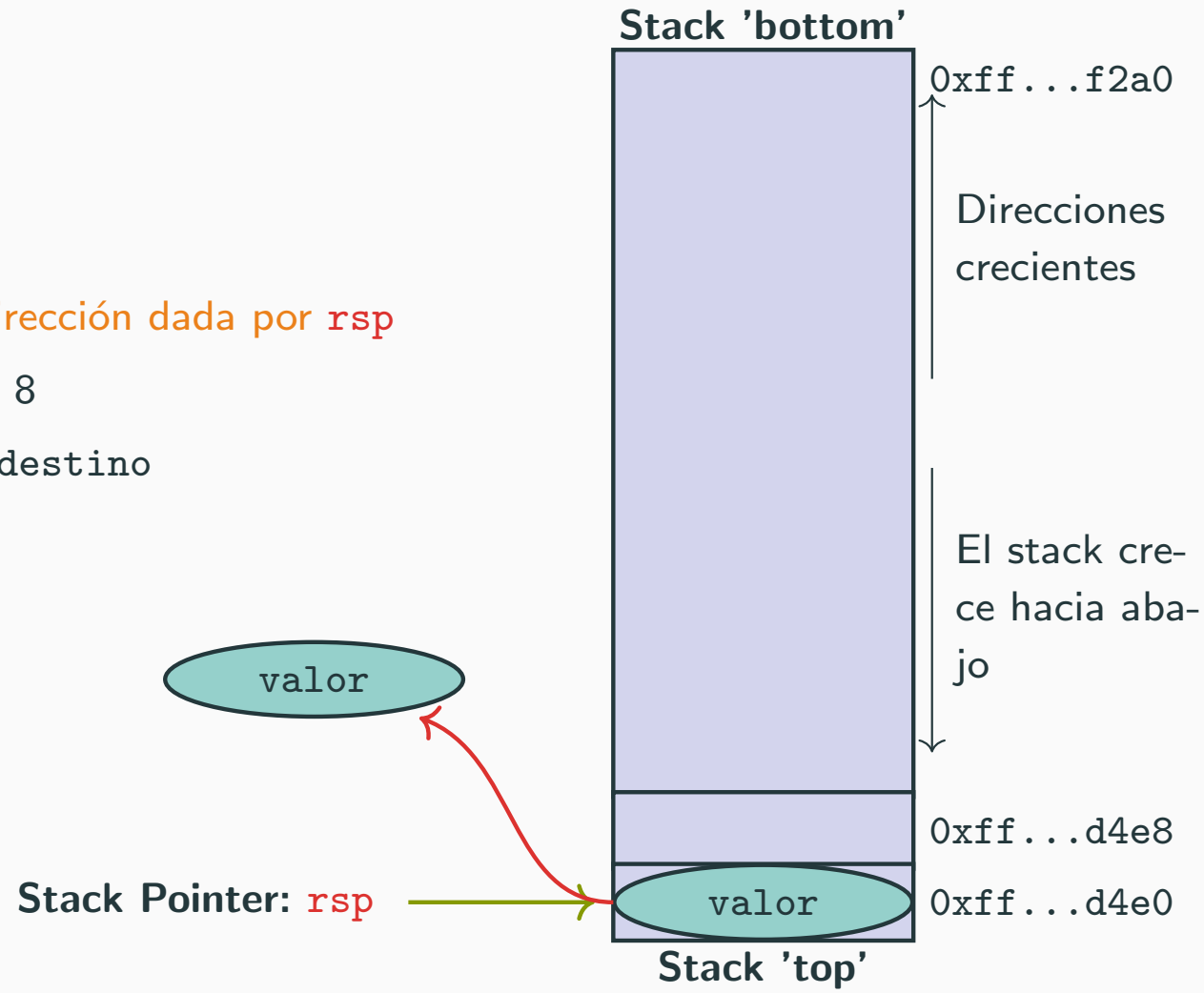
- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`



Stack x86-64: pop

`pop destino`

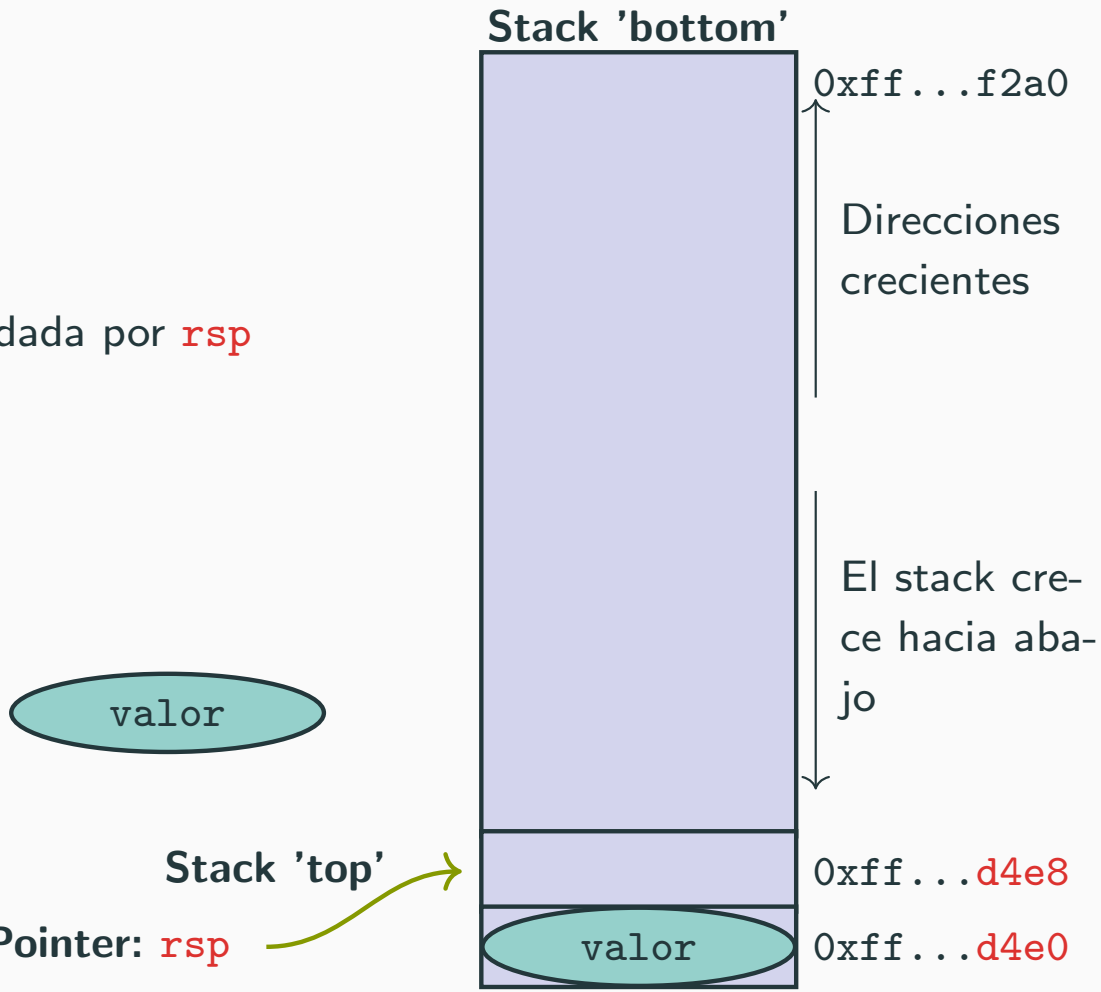
- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`



Stack x86-64: pop

`pop destino`

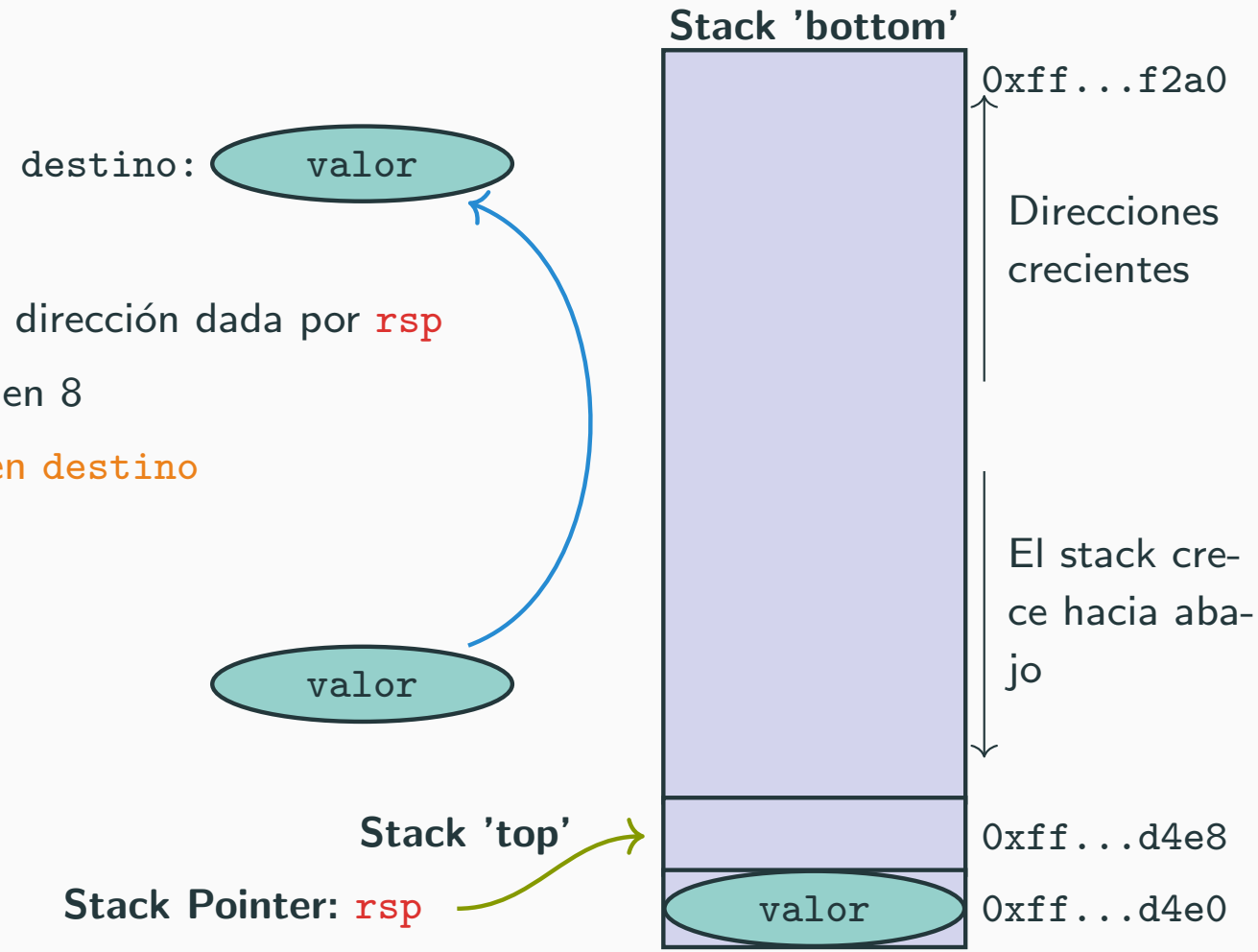
- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`



Stack x86-64: pop

`pop destino`

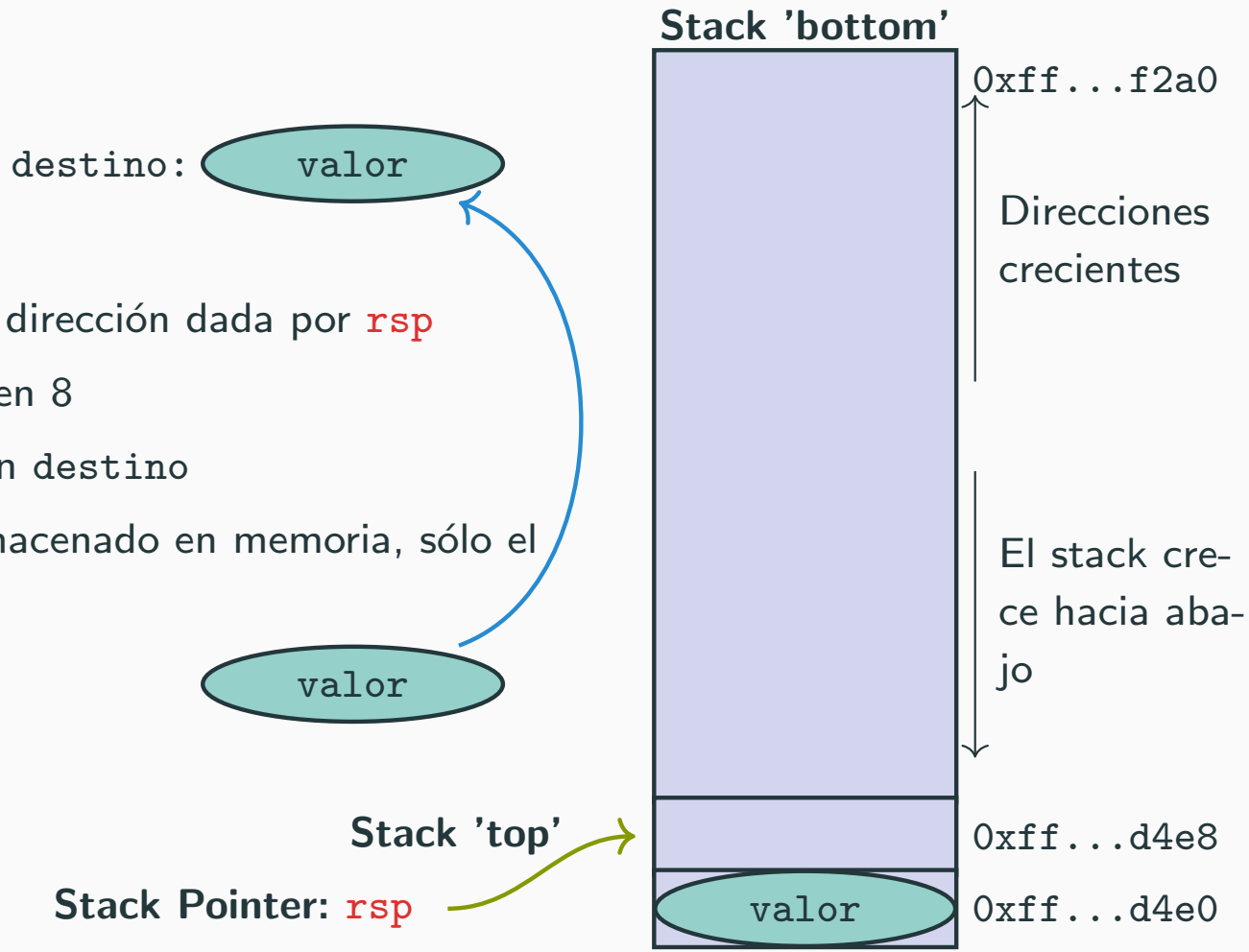
- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`



Stack x86-64: pop

`pop destino`

- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`
- No cambia lo almacenado en memoria, sólo el valor de `rsp`



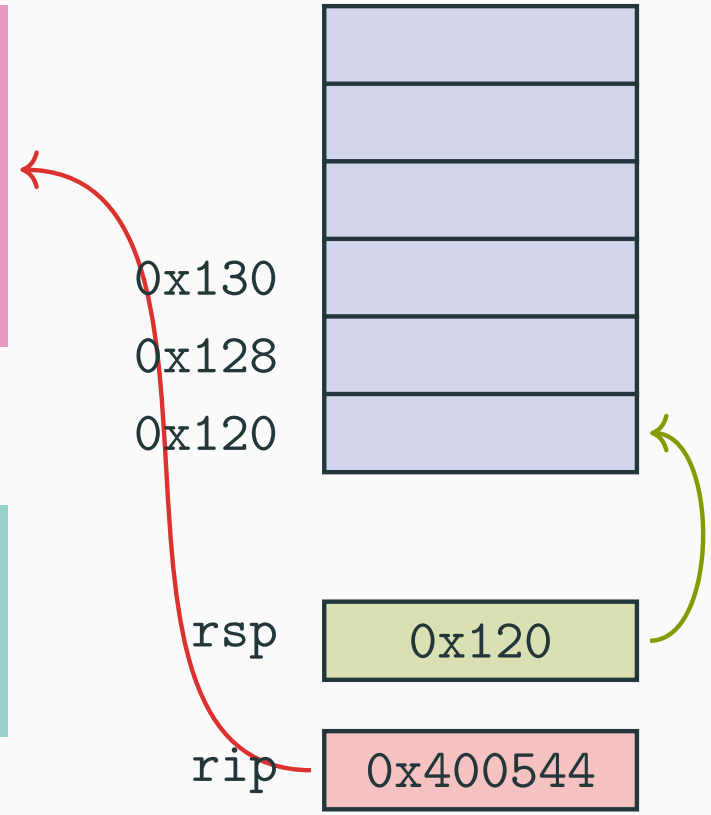
Transferencia de control

- Usa el *stack* para dar soporte a las llamadas y retornos de procedimientos
- Llamada a procedimientos/funciones: `call etiqueta`
  - Hacer un `push` de la *dirección de retorno*
  - "Saltar" a la `etiqueta`
- *Dirección de retorno*
  - Dirección de la instrucción siguiente (inmediata) a la instrucción `call`
- Retorno de procedimientos/funciones: `ret`
  - Hacer un `pop` de la dirección de retorno
  - "Saltar" a dicha dirección

Transferencia de control: ejemplo


```
0000000000400540 <multstore>:
   ...
   ...
   400544: call 400550 <mult2>
   400549: mov   [rbx], rax
   ...
   ...
```

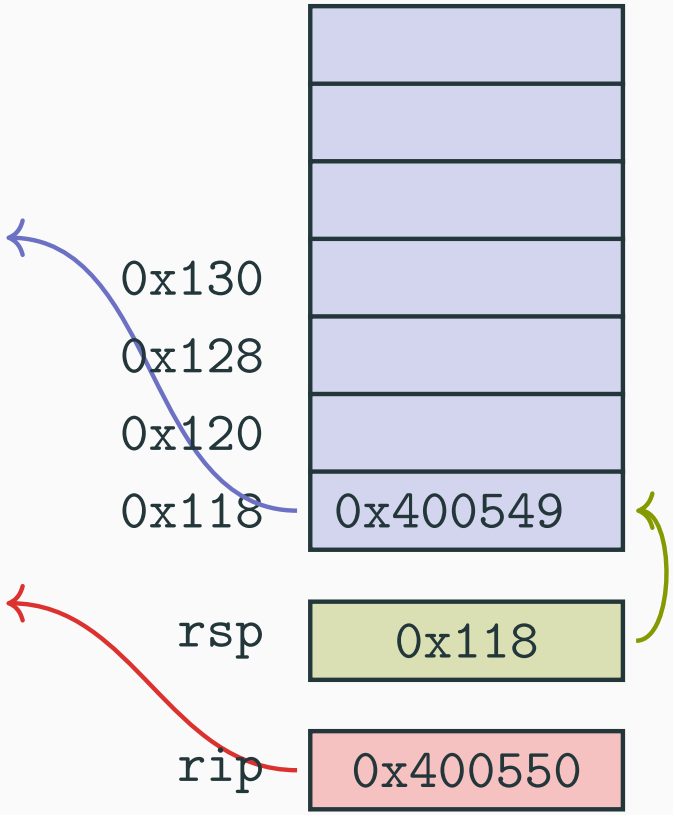
```
0000000000400550 <mult2>:
   400550: mov   rax, rdi
   ...
   400557: ret
```



Transferencia de control: ejemplo


```
0000000000400540 <multstore>:
   ...
   ...
   400544: call 400550 <mult2>
   400549: mov   [rbx], rax
   ...
   ...
```

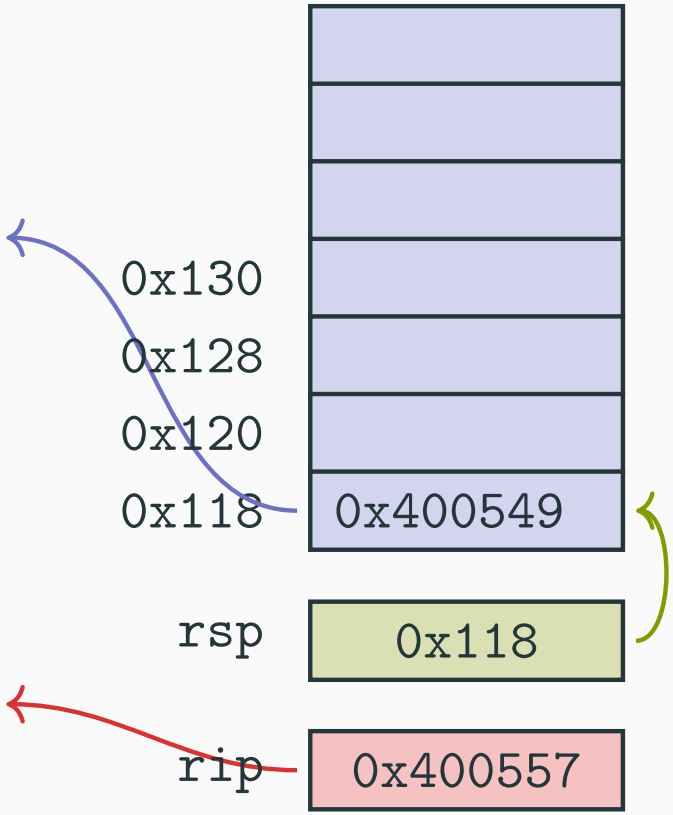
```
0000000000400550 <mult2>:
   400550: mov   rax, rdi
   ...
   400557: ret
```



Transferencia de control: ejemplo


```
0000000000400540 <multstore>:
   ...
   ...
   400544: call 400550 <mult2>
   400549: mov   [rbx], rax
   ...
   ...
```

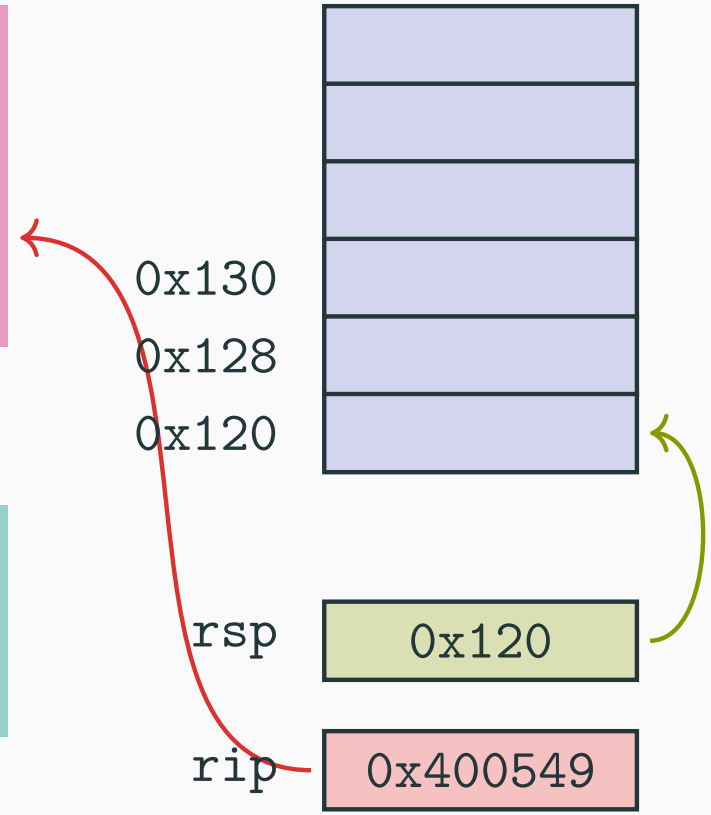
```
0000000000400550 <mult2>:
   400550: mov   rax, rdi
   ...
   400557: ret
```



Transferencia de control: ejemplo


```
0000000000400540 <multstore>:
   ...
   ...
   400544: call 400550 <mult2>
   400549: mov   [rbx], rax
   ...
   ...
```

```
0000000000400550 <mult2>:
   400550: mov   rax, rdi
   ...
   400557: ret
```



Pasaje de parámetros

- Los argumentos se pasan por registros o usando el stack
  - en x86 (32 bits) únicamente usando la pila

- **Primeros 6 argumentos**

`rdi`	arg 1
`rsi`	arg 2
`rdx`	arg 3
`rsx`	arg 4
`r8`	arg 5
`r9`	arg 6

- **Valor de retorno**

`rax`

- **Argumentos siguientes**

Stack bottom

`...`
`arg n`
`...`
`arg 8`
`arg 7`

Stack top

- El espacio para los argumentos se reserva únicamente si es necesario

Ejemplo de cadena de invocaciones

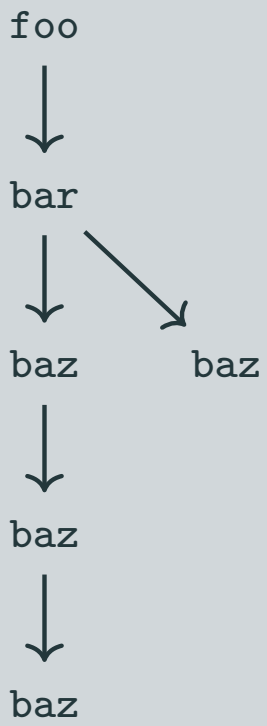

```
foo (...)
{
  ...
  ...
bar (...);
  ...
  ...
}
```

```
bar (...)
{
  ...
baz (...);
  ...
baz (...);
  ...
}
```

```
baz (...)
{
  ...
  ...
baz (...);
  ...
  ...
}
```

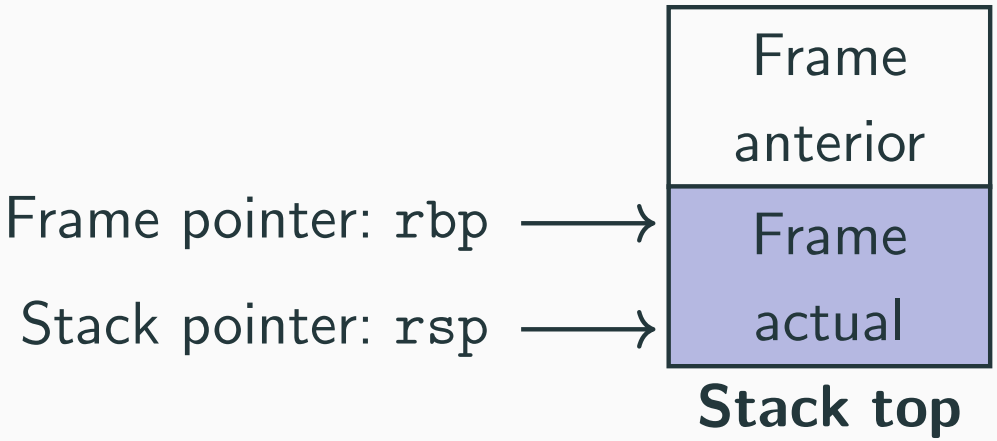
baz() es recursivo

Árbol de llamadas



Stack frames

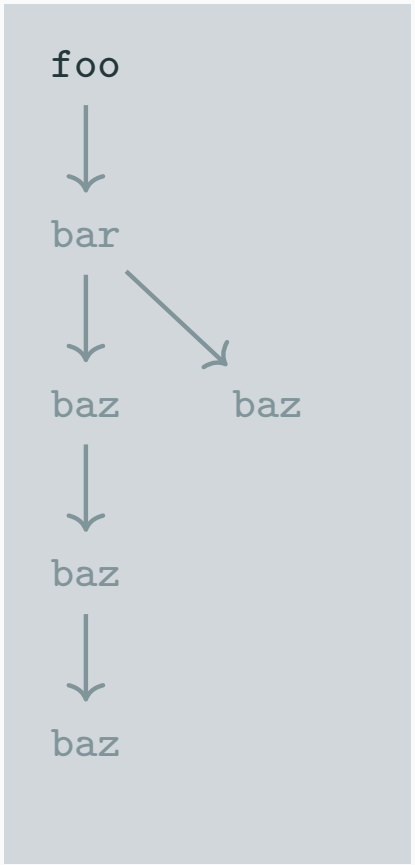
- **Contiene:**
  - Información de retorno
  - Almacenamiento local
  - Espacio temporal

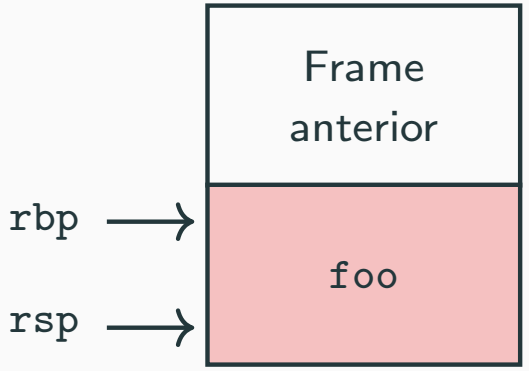


- **Administración:**
  - El espacio se reserva al entrar
    - Requiere código de inicialización
    - Incluye el `push` de la instrucción `call`
  - El espacio se retorna al salir
    - Requiere código de finalización
    - Incluye el `pop` de la instrucción `ret`

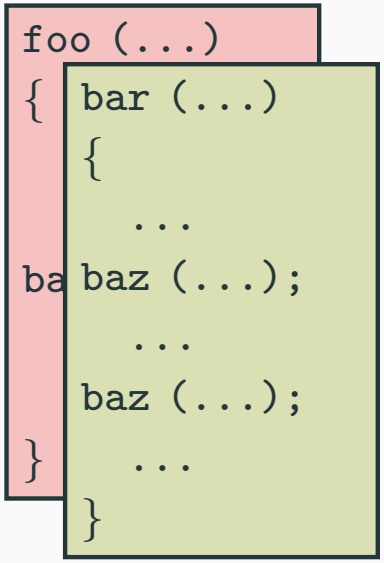
Stack frames: ejemplo

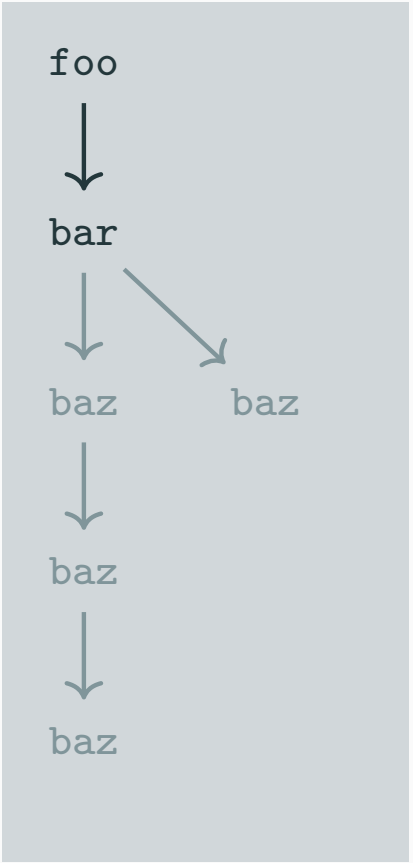

```
foo (...)
{
  ...
  ...
bar (...);
  ...
  ...
}
```

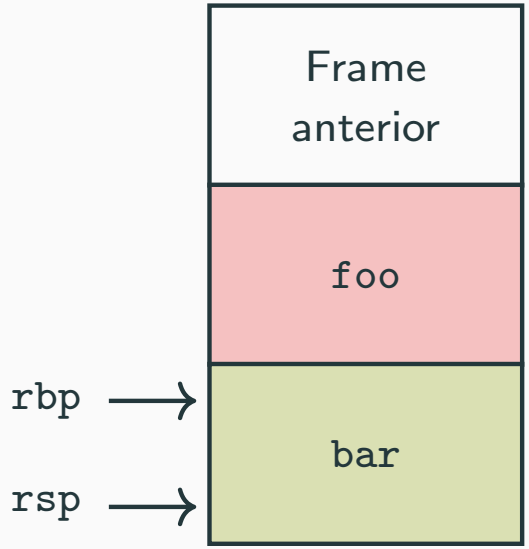




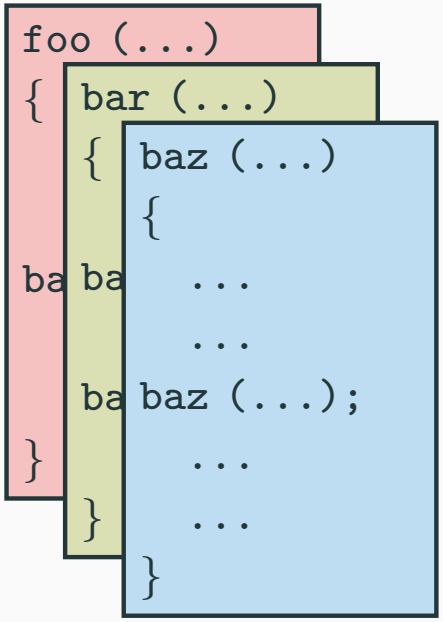
Stack frames: ejemplo

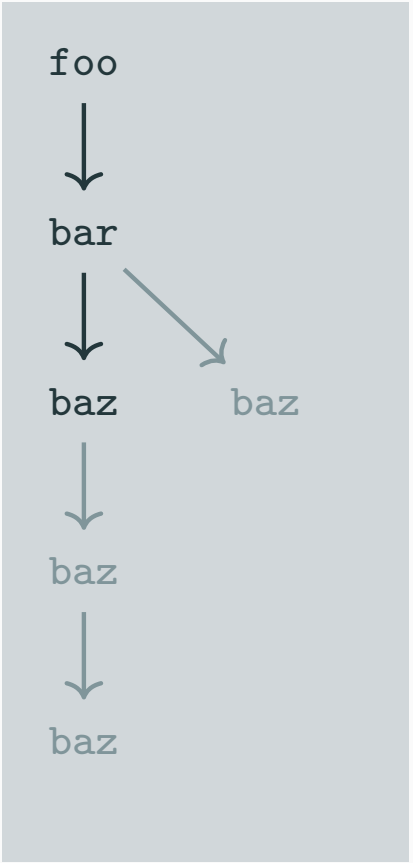


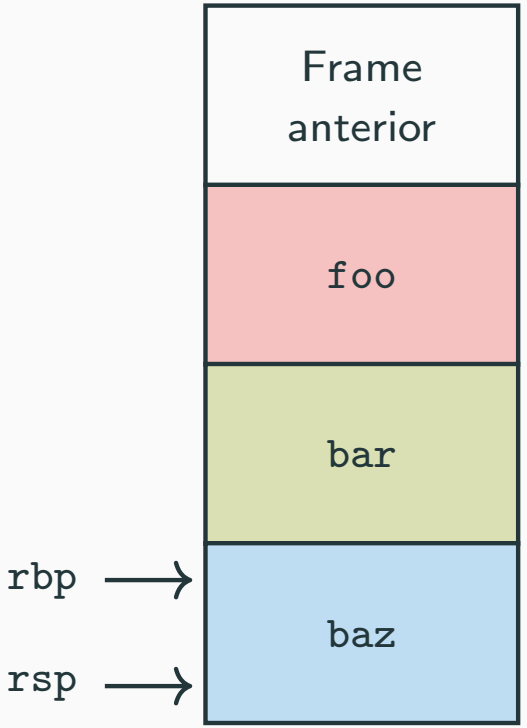




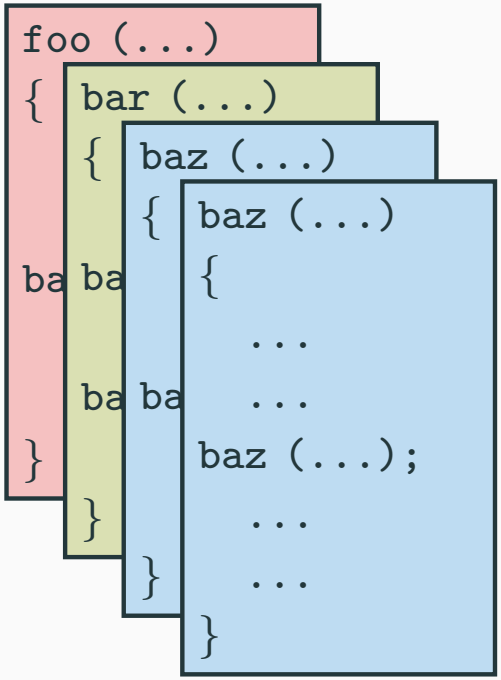
Stack frames: ejemplo

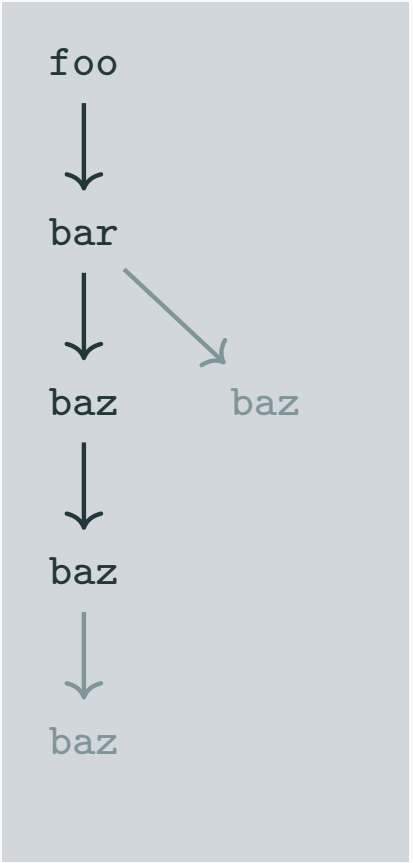


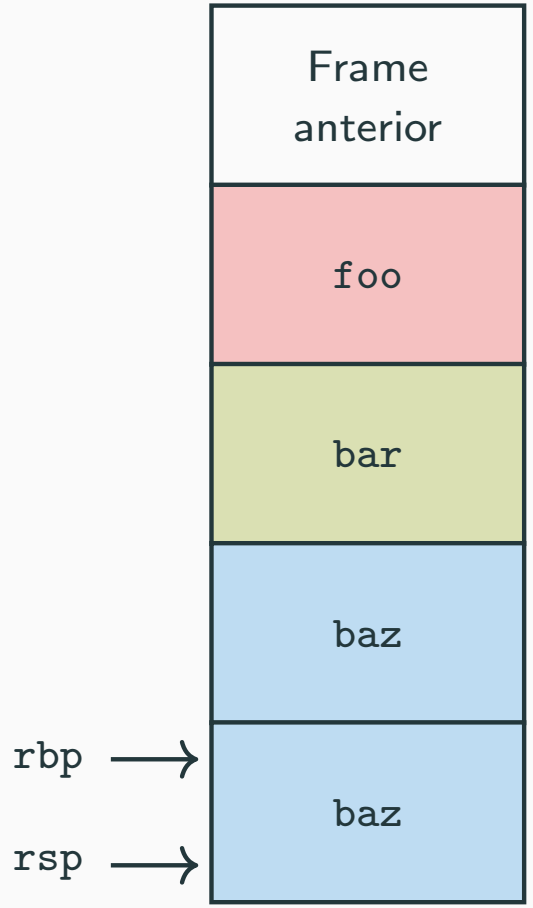




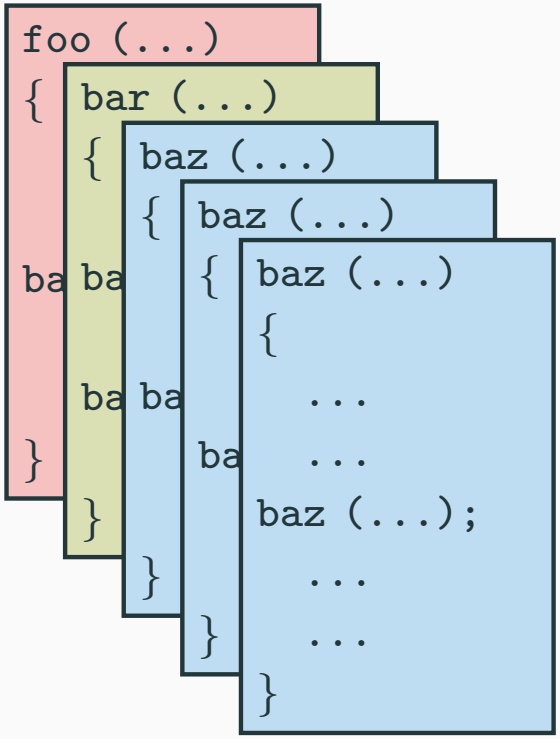
Stack frames: ejemplo

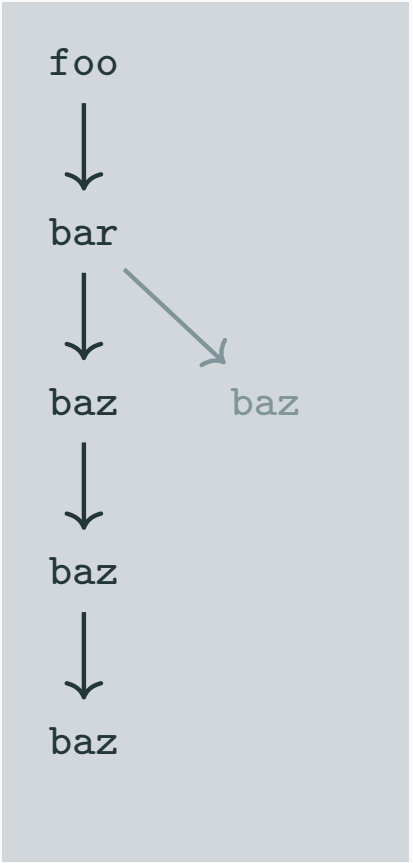


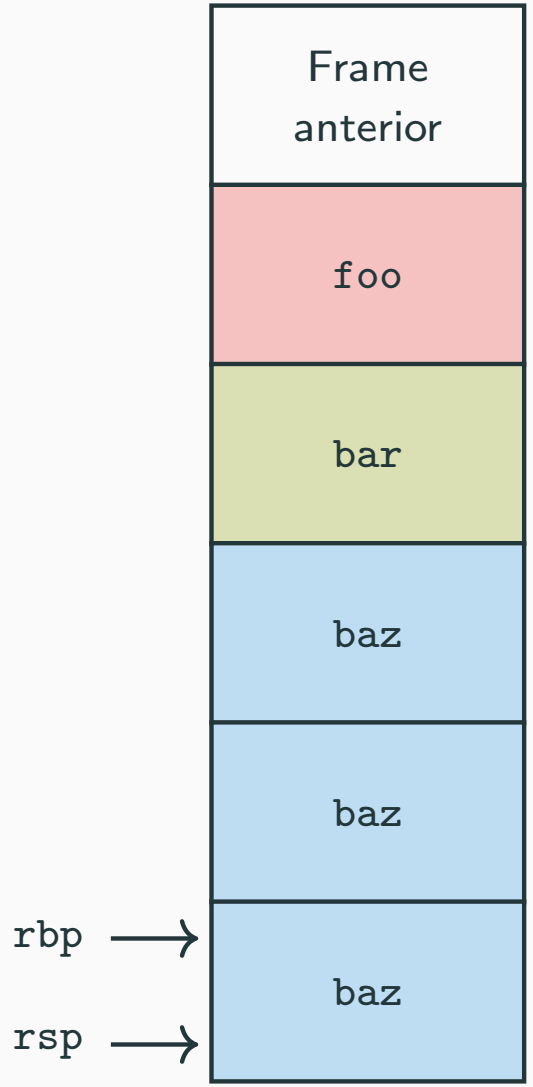




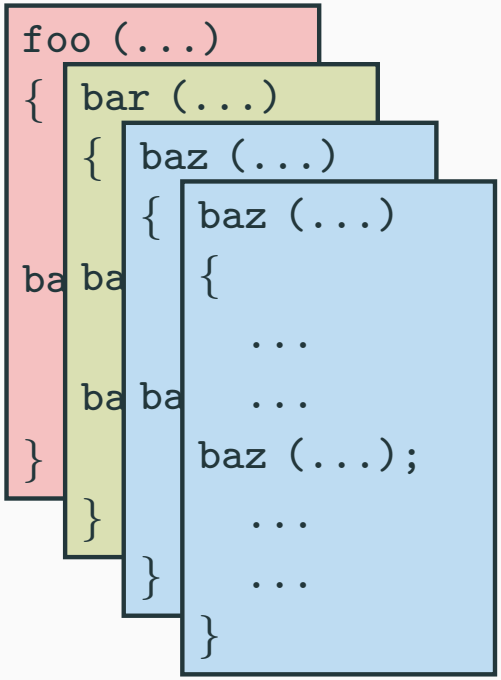
Stack frames: ejemplo

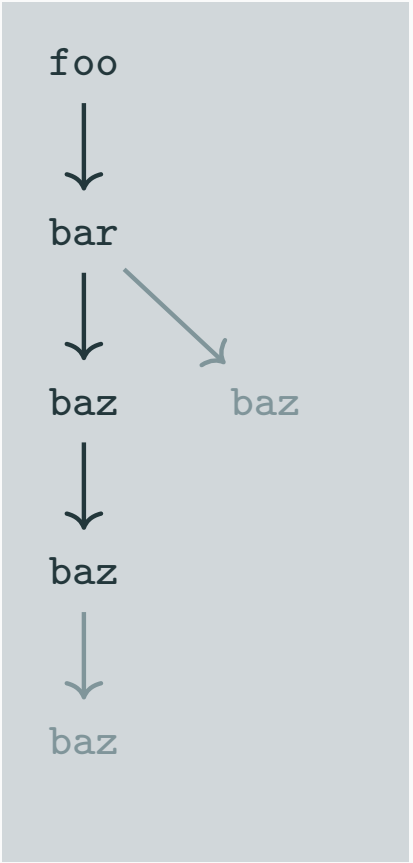


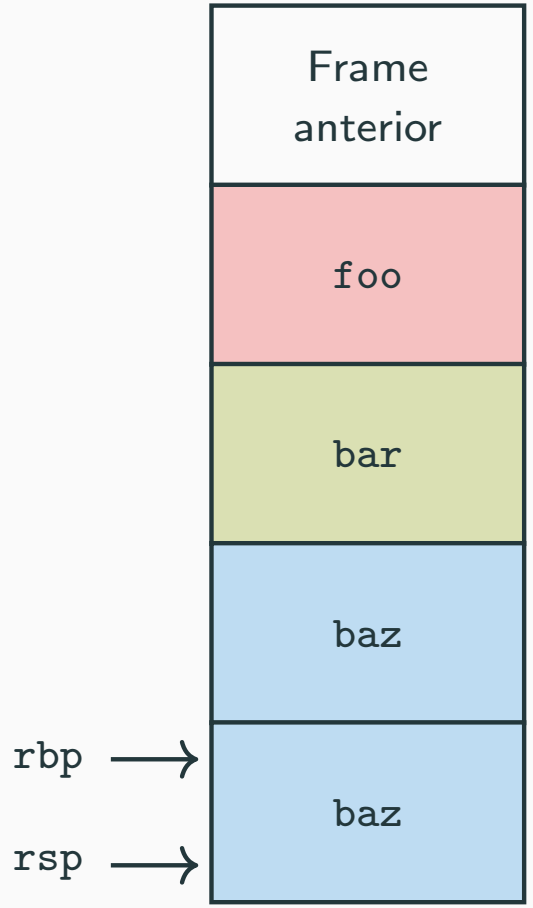




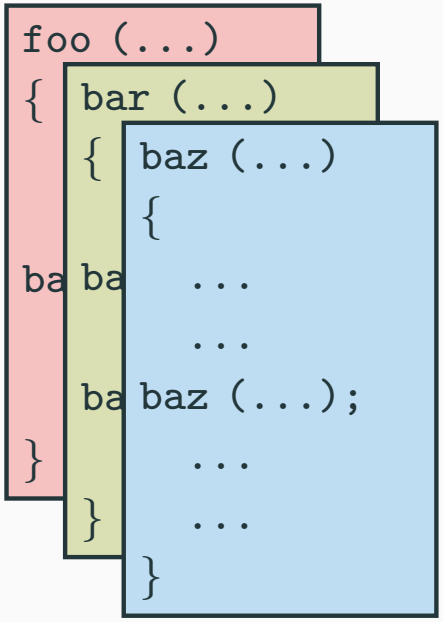
Stack frames: ejemplo

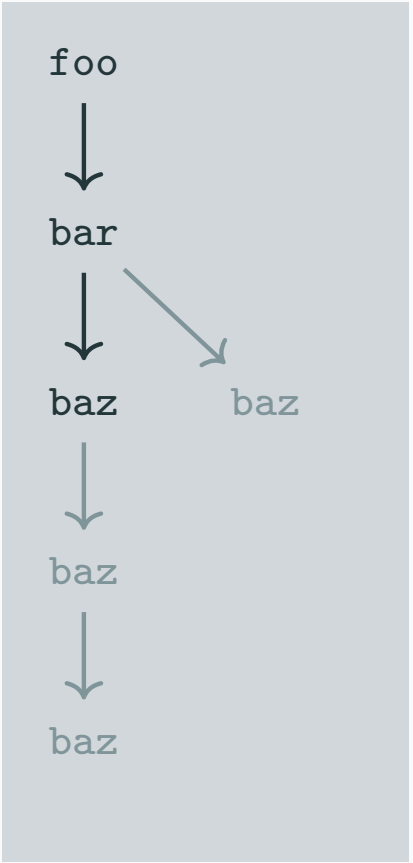


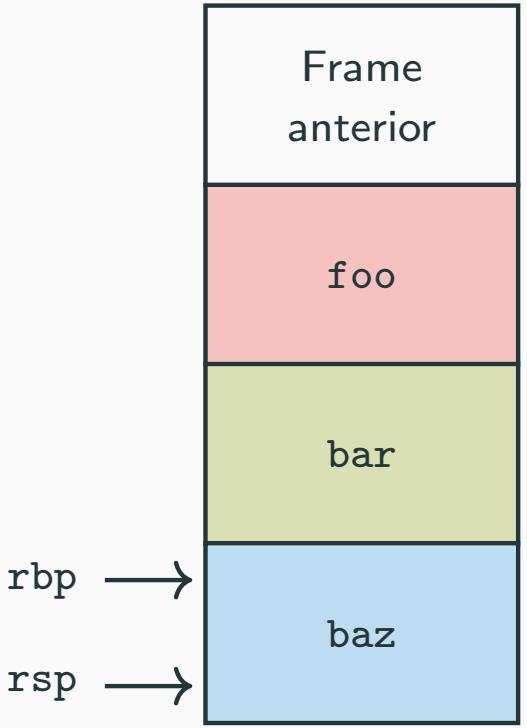




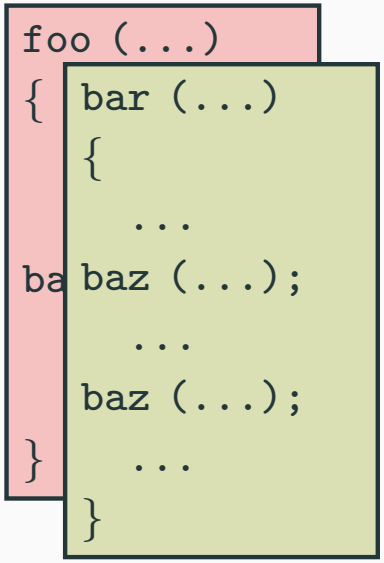
Stack frames: ejemplo

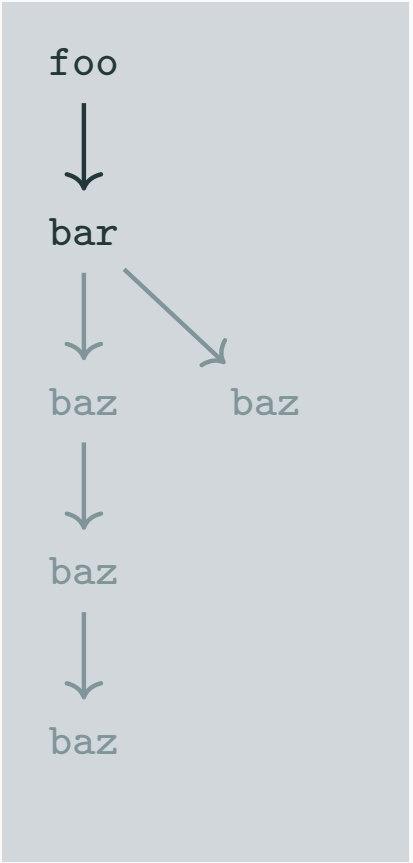


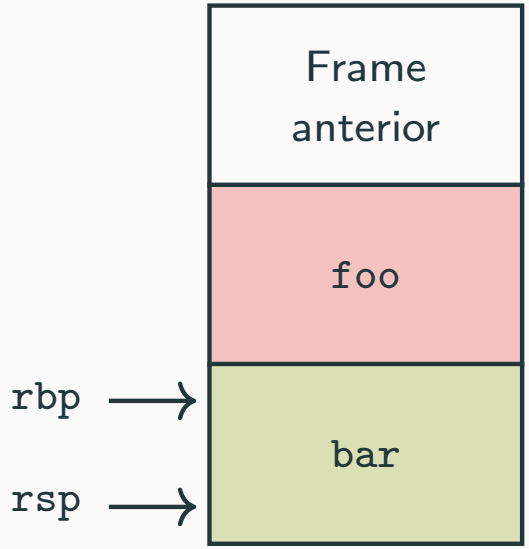




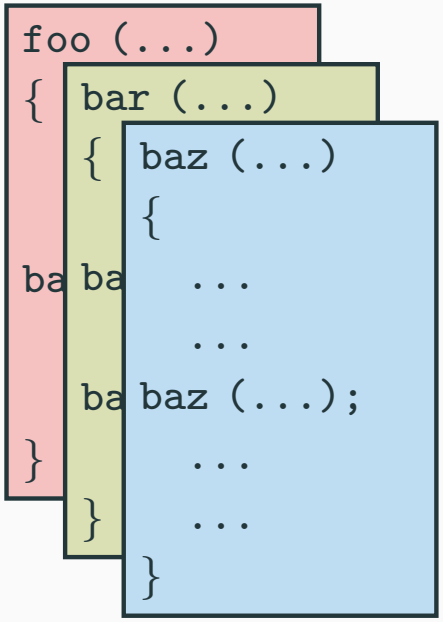
Stack frames: ejemplo

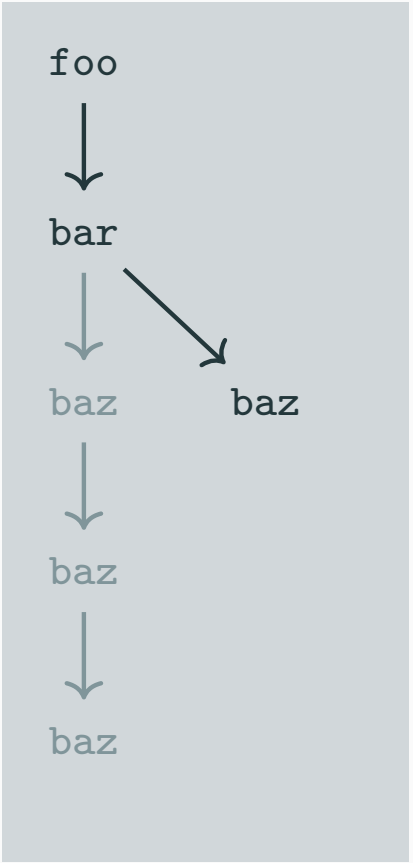


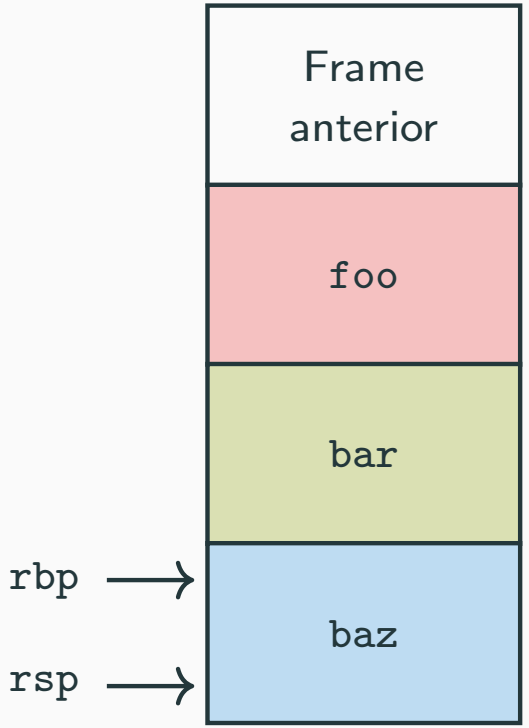




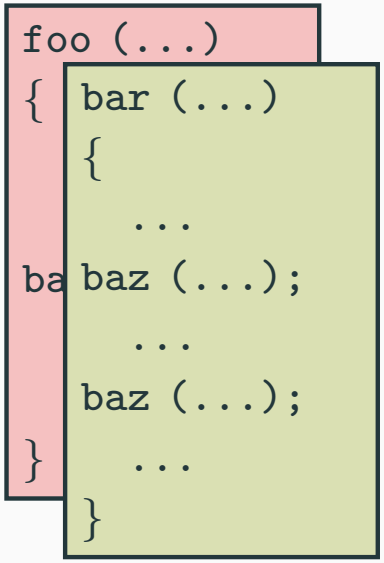
Stack frames: ejemplo

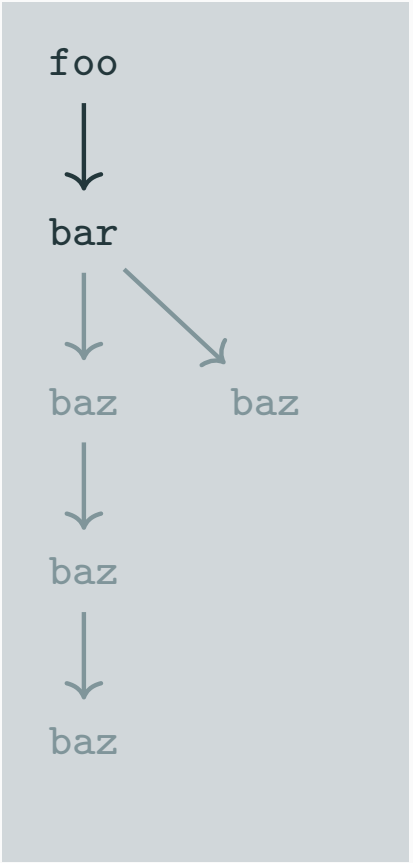


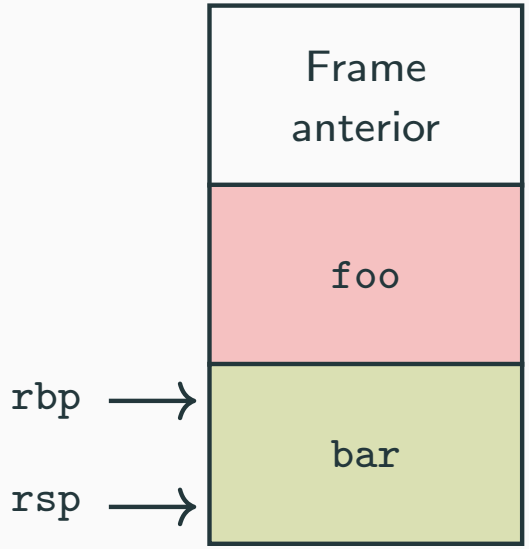




Stack frames: ejemplo

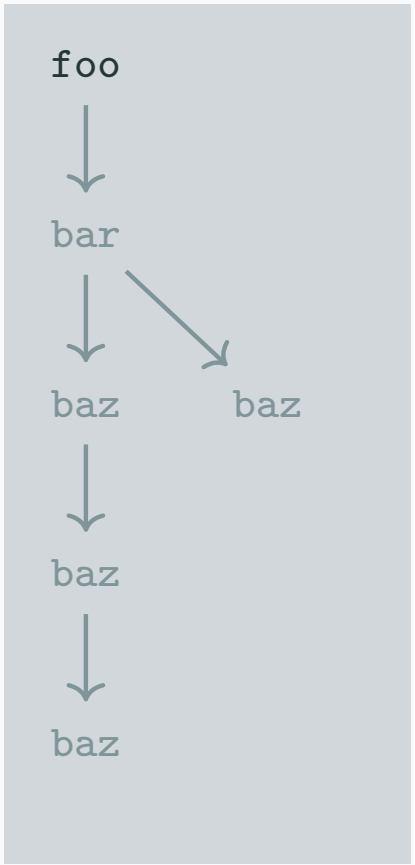


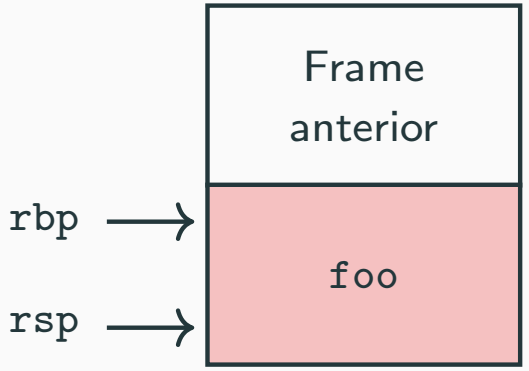




Stack frames: ejemplo

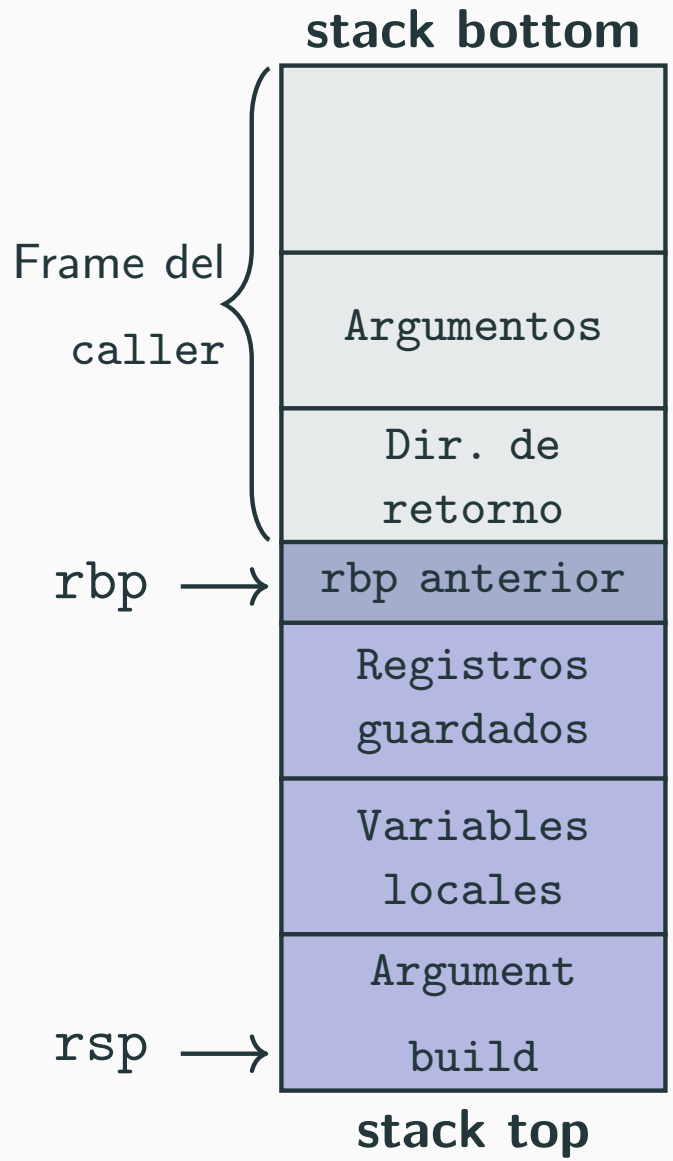

```
foo (...)
{
  ...
  ...
bar (...);
  ...
  ...
}
```





Linux Stack Frame

- **Frame actual (top a bottom)**
  - *argument build*: parámetros de una función a ser invocada
  - variables locales (si no alcanzan los registros)
  - Registros guardados
  - *frame pointer* anterior
- **Frame de la función invocante**
  - dirección de retorno
    - `pusheada` por `call`
  - argumentos para esta función
    - En x86_64 (64 bits): del séptimo en adelante
    - En x86 (32 bits): todos



Convenciones para registros

Pregunta: ¿Qué pasa con los registros en uso al realizar una llamada? ¿Conservan sus valores al regresar?

Convenciones para registros

Pregunta: ¿Qué pasa con los registros en uso al realizar una llamada? ¿Conservan sus valores al regresar?

Respuesta: Vamos a dividir los registros entre volátiles y no volátiles

Convenciones para registros

Pregunta: ¿Qué pasa con los registros en uso al realizar una llamada? ¿Conservan sus valores al regresar?

Respuesta: Vamos a dividir los registros entre volátiles y no volátiles

- **Volátiles**: La función llamada no tiene obligación de conservarlos (**Caller-saved**).

Convenciones para registros

Pregunta: ¿Qué pasa con los registros en uso al realizar una llamada? ¿Conservan sus valores al regresar?

Respuesta: Vamos a dividir los registros entre volátiles y no volátiles

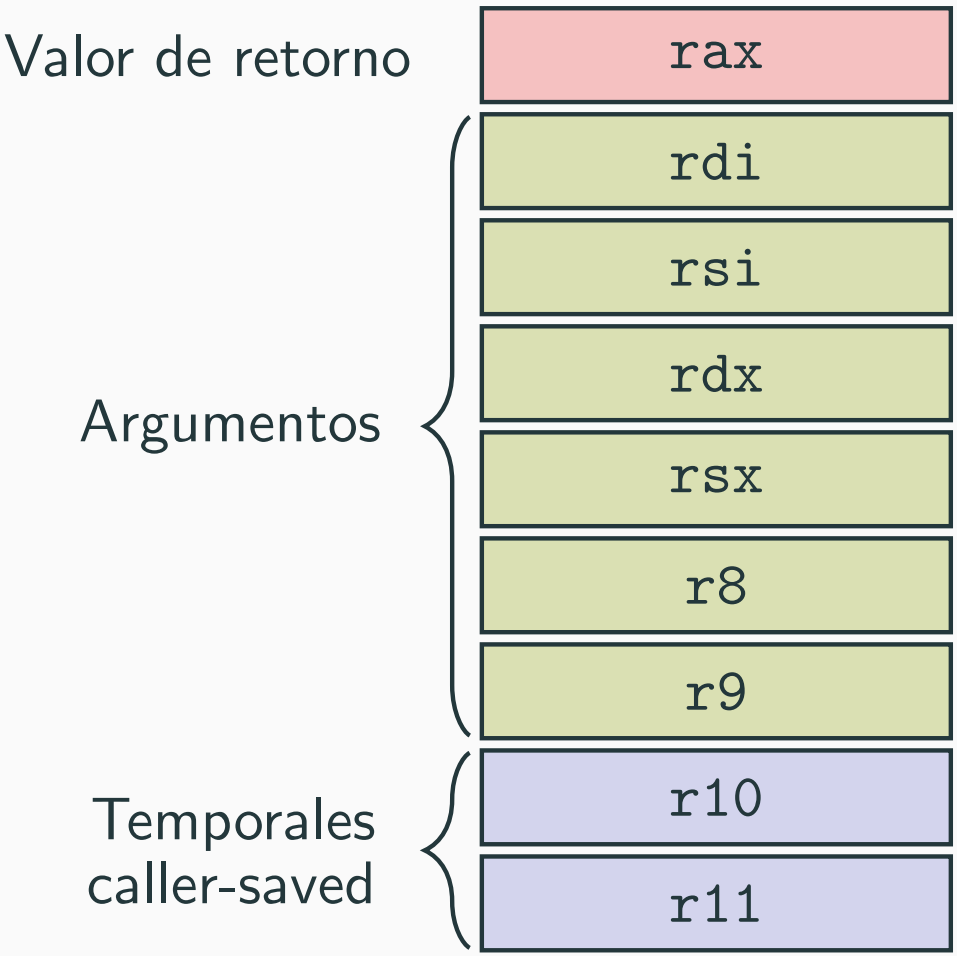
- **Volátiles**: La función llamada no tiene obligación de conservarlos (**Caller-saved**).
- **No volátiles**: Si la función llamada los cambia debe restaurarlos antes del `return` (**Callee-saved**).

Convenciones para registros

- **Cuando `foo` llama a `bar`:**
  - `foo` se llama **caller** o invocante
  - `bar` se llama **callee** o invocada
- **¿Qué ocurre con los registros usados como temporales?**
  - El contenido de éstos puede ser sobreescrito por la función **callee**
  - Es necesario algún arreglo de partes para que funcione correctamente
- **Convenciones (calling conventions):**
  - Caller saved
    - la función *caller* guarda los registros en el stack antes de la invocación
  - Callee saved
    - la función *callee* guarda los registros en el stack antes de **modificarlos**
    - la función *callee* reestable los valores de los registros modificados antes de retornar

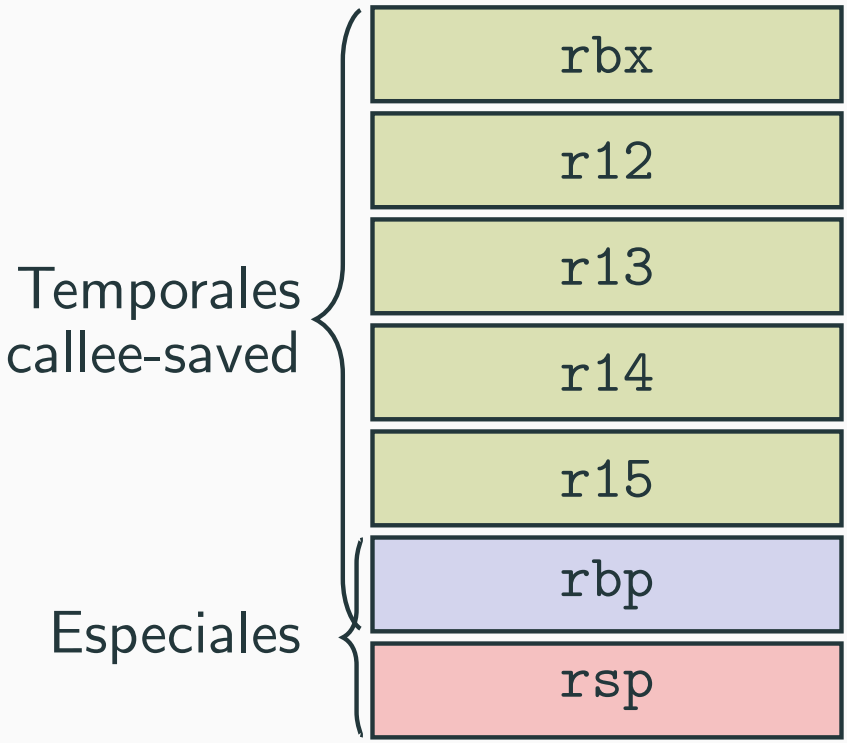
Convenciones para registros caller-saved en x86_64

- `rax`
  - valor de retorno
  - caller-saved
  - un procedimiento puede modificarlo
- `rdi, ..., r9`
  - argumentos
  - caller-saved
  - un procedimiento puede modificarlos
- `r10, r11`
  - caller-saved
  - un procedimiento puede modificarlos



Convenciones para registros callee-saved en x86_64

- `rbx`, `r12`, `r13`, `r14`, `r15`
  - callee-saved
  - la función callee debe guardarlos y restaurarlos
- `rbp`
  - callee-saved
  - la función callee debe guardarlos y restaurarlos
  - opcionalmente puede usarse como frame pointer
- `rsp`
  - callee-saved especial
  - se restaura a su valor original al retornar del procedimiento



Resumen System V

System V - Versión de 64 bits

En 64 bits `System V` define lo siguiente:

[1] Para el ABI los punteros son enteros ¯_(ツ)_/¯

System V - Versión de 64 bits

En 64 bits `System V` define lo siguiente:

- Los registros `RBX`, `RBP`, `R12`, `R13`, `R14` y `R15` son **no volátiles**

[1] Para el ABI los punteros son enteros ¯_(ツ)_/¯

System V - Versión de 64 bits

En 64 bits `System V` define lo siguiente:

- Los registros `RBX`, `RBP`, `R12`, `R13`, `R14` y `R15` son **no volátiles**
- El valor de retorno será almacenado en `RAX` para valores enteros[1] y en `XMM0` para flotantes

[1] Para el ABI los punteros son enteros ¯_(ツ)_/¯

System V - Versión de 64 bits

En 64 bits `System V` define lo siguiente:

- Los registros `RBX`, `RBP`, `R12`, `R13`, `R14` y `R15` son **no volátiles**
- El valor de retorno será almacenado en `RAX` para valores enteros[1] y en `XMM0` para flotantes
- Al salir de la función llamada la pila debe encontrarse en el mismo estado en el que estaba al ingresar (todo `PUSH` debe tener su `POP`)(¿RSP volátil?)

[1] Para el ABI los punteros son enteros ¯_(ツ)_/¯

System V - Versión de 64 bits

En 64 bits `System V` define lo siguiente:

- Los registros `RBX`, `RBP`, `R12`, `R13`, `R14` y `R15` son **no volátiles**
- El valor de retorno será almacenado en `RAX` para valores enteros[1] y en `XMM0` para flotantes
- Al salir de la función llamada la pila debe encontrarse en el mismo estado en el que estaba al ingresar (todo `PUSH` debe tener su `POP`)(¿RSP volátil?)
- Antes de realizar una llamada a una función, la pila debe estar alineada a 16 bytes

[1] Para el ABI los punteros son enteros ¯_(ツ)_/¯

System V - Versión de 64 bits

Aclaración: Donde diga **de derecha a izquierda** o **de izquierda a derecha** debemos entender que nos referimos al orden de los parámetros en la declaración de la función en el encabezado `.h`.

System V - Versión de 64 bits

En 64 bits la `System V` define lo siguiente:

System V - Versión de 64 bits

En 64 bits la `System V` define lo siguiente:

- Los parámetros enteros se pasan de izquierda a derecha en `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` respectivamente

System V - Versión de 64 bits

En 64 bits la `System V` define lo siguiente:

- Los parámetros enteros se pasan de izquierda a derecha en `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` respectivamente
- Los parámetros flotantes se pasan de izquierda a derecha en `XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6`, `XMM7` respectivamente

System V - Versión de 64 bits

En 64 bits la `System V` define lo siguiente:

- Los parámetros enteros se pasan de izquierda a derecha en `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` respectivamente
- Los parámetros flotantes se pasan de izquierda a derecha en `XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6`, `XMM7` respectivamente
- Si no hay registros disponibles para los parámetros enteros y/o flotantes se pasarán de **derecha a izquierda** a través de la pila haciendo `PUSH`.

System V - Versión de 64 bits

En 64 bits la `System V` define lo siguiente:

- Los parámetros enteros se pasan de izquierda a derecha en `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` respectivamente
- Los parámetros flotantes se pasan de izquierda a derecha en `XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6`, `XMM7` respectivamente
- Si no hay registros disponibles para los parámetros enteros y/o flotantes se pasarán de **derecha a izquierda** a través de la pila haciendo `PUSH`.
- Las estructuras se tratan de una forma especial (ver referencia). Si son grandes se pasa un puntero a la misma como parámetro.

System V - Versión de 64 bits

En 64 bits la `System V` define lo siguiente:

- Los parámetros enteros se pasan de izquierda a derecha en `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` respectivamente
- Los parámetros flotantes se pasan de izquierda a derecha en `XMM0`, `XMM1`, `XMM2`, `XMM3`, `XMM4`, `XMM5`, `XMM6`, `XMM7` respectivamente
- Si no hay registros disponibles para los parámetros enteros y/o flotantes se pasarán de **derecha a izquierda** a través de la pila haciendo `PUSH`.
- Las estructuras se tratan de una forma especial (ver referencia). Si son grandes se pasa un puntero a la misma como parámetro.
- Si el valor a retornar es una estructura grande la función llamadora reserva espacio (parámetro implícito).

System V - Versión de 64 bits (machete)

Para que tengan a mano en la **primera parte de la materia (64 bits)**:

No volátiles	`RBX`, `RBP`, `R12`, `R13`, `R14` y `R15` (¿`RSP`?)
Valor de retorno	`RAX`enteros/punteros, `XMM0` flotantes
Entero, puntero	`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` (izq. a der.)
Flotantes	`XMM0`, `XMM1`, . . . , `XMM7` (izq. a der.)
¿No hay registros?	`PUSH` a la pila (der. a izq.)
Inv. de pila	Todo `PUSH`/`SUB` debe tener su `POP`/`ADD`
Llamada a func.	Pila alineada a 16 bytes

System V - Versión de 32 bits

En 32 bits la `System V` define lo siguiente:

System V - Versión de 32 bits

En 32 bits la `System V` define lo siguiente:

- Los registros `EBX`, `EBP`, `ESI` y `EDI` son **no volátiles**.

System V - Versión de 32 bits

En 32 bits la `System V` define lo siguiente:

- Los registros `EBX`, `EBP`, `ESI` y `EDI` son **no volátiles**.
- El valor de retorno será almacenado en `EAX`.

System V - Versión de 32 bits

En 32 bits la `System V` define lo siguiente:

- Los registros `EBX`, `EBP`, `ESI` y `EDI` son **no volátiles**.
- El valor de retorno será almacenado en `EAX`.
- Al salir de la función llamada la pila debe encontrarse en el mismo estado en el que estaba al ingresar (todo `PUSH` debe tener su `POP`).

System V - Versión de 32 bits

En 32 bits la `System V` define lo siguiente:

- Los registros `EBX`, `EBP`, `ESI` y `EDI` son **no volátiles**.
- El valor de retorno será almacenado en `EAX`.
- Al salir de la función llamada la pila debe encontrarse en el mismo estado en el que estaba al ingresar (todo `PUSH` debe tener su `POP`).
- Los parámetros se pasarán de **derecha a izquierda** a través de la pila haciendo `PUSH` (¿ESP volátil?.)

System V - Versión de 32 bits

En 32 bits la `System V` define lo siguiente:

- Los registros `EBX`, `EBP`, `ESI` y `EDI` son **no volátiles**.
- El valor de retorno será almacenado en `EAX`.
- Al salir de la función llamada la pila debe encontrarse en el mismo estado en el que estaba al ingresar (todo `PUSH` debe tener su `POP`).
- Los parámetros se pasarán de **derecha a izquierda** a través de la pila haciendo `PUSH` (¿ESP volátil?.)
- Antes de realizar una llamada a una función la pila debe quedar alineada a 16 bytes.

System V - Versión de 32 bits

En 32 bits la `System V` define lo siguiente:

- Los registros `EBX`, `EBP`, `ESI` y `EDI` son **no volátiles**.
- El valor de retorno será almacenado en `EAX`.
- Al salir de la función llamada la pila debe encontrarse en el mismo estado en el que estaba al ingresar (todo `PUSH` debe tener su `POP`).
- Los parámetros se pasarán de **derecha a izquierda** a través de la pila haciendo `PUSH` (¿ESP volátil?.)
- Antes de realizar una llamada a una función la pila debe quedar alineada a 16 bytes.
- Si el valor a retornar es una estructura grande la función llamadora reserva espacio (parámetro implícito).

System V - Versión de 32 bits (machete)

Para que tengan a mano en la **segunda parte de la materia (32 bits)**:

No volátiles	`EBP`, `EBX`, `ESI` y `EDI` (¿`ESP`?)
Valor de retorno	`EAX`
Parámetros	`PUSH` a la pila (der. a izq.)
Inv. de pila	Todo `PUSH`/`SUB` debe tener su `POP`/`ADD`
Llamada a func	Pila alineada a 16 bytes

Pregunta: ¿Por qué hace falta alinear la pila a 16 bytes si hacemos una llamada a otra biblioteca?

Pregunta: ¿Por qué hace falta alinear la pila a 16 bytes si hacemos una llamada a otra biblioteca?

Respuesta: Las funciones pueden hacer uso de operaciones de registros largos (`XMM`, `YMM`) que requieren datos alineados a 16 bytes, es por esto que el contrato de uso de un conjunto de instrucciones del procesador se traduce en un contrato de uso de nuestras funciones de bajo nivel.

Consultas