

# Lenguaje de máquina: procedimientos

95.57/75.03 Organización del computador

---

**Docentes:** Patricio Moreno y Adeodato Simó

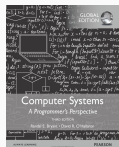
2.<sup>do</sup> cuatrimestre de 2020

Última modificación: Tue Aug 4 14:36:17 2020 -0300

Facultad de Ingeniería (UBA)

# Créditos

Para armar las presentaciones del curso nos basamos en:



R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.



D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.



J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

# Tabla de contenidos

---

## 1. Procedimientos

Mecanismos necesarios

Pila (*Stack*)

Calling conventions

Transferencia de control

Pasaje de datos

Manejo de datos locales

# Tabla de contenidos

---

## 1. Procedimientos

Mecanismos necesarios

Pila (*Stack*)

Calling conventions

Transferencia de control

Pasaje de datos

Manejo de datos locales

# Tabla de contenidos

---

## 1. Procedimientos

Mecanismos necesarios

Pila (*Stack*)

Calling conventions

Transferencia de control

Pasaje de datos

Manejo de datos locales

# Soporte para procedimientos

- Transferencia del control
  - Necesario para “saltar” al código del procedimiento
  - Retorno al punto del salto
- Pasaje de datos
  - Argumentos
  - Valor de retorno
- Manejo de la memoria
  - Reservar lo necesario en el procedimiento
  - Liberar lo pedido al retornar
- Todos los mecanismos se implementan a través de instrucciones
- Todo procedimiento en x86-64 utiliza esos mecanismos

```
P(...) {  
    .  
    .  
    y = Q(x);  
    print(y)  
}
```

```
int Q(int i) {  
    int t = 3 * i;  
    int v[10];  
    .  
    .  
    return v[0];  
}
```

# Soporte para procedimientos

- Transferencia del control
  - Necesario para “saltar” al código del procedimiento
  - Retorno al punto del salto
- Pasaje de datos
  - Argumentos
  - Valor de retorno
- Manejo de la memoria
  - Reservar lo necesario en el procedimiento
  - Liberar lo pedido al retornar
- Todos los mecanismos se implementan a través de instrucciones
- Todo procedimiento en x86-64 utiliza esos mecanismos

```
P(...) {  
    .  
    .  
    y = Q(x);  
    print(y)  
}  
  
int Q(int i) {  
    int t = 3 * i;  
    int v[10];  
    .  
    .  
    return v[0];  
}
```

# Soporte para procedimientos

- Transferencia del control
  - Necesario para “saltar” al código del procedimiento
  - Retorno al punto del salto
- Pasaje de datos
  - Argumentos
  - Valor de retorno
- Manejo de la memoria
  - Reservar lo necesario en el procedimiento
  - Liberar lo pedido al retornar
- Todos los mecanismos se implementan a través de instrucciones
- Todo procedimiento en x86-64 utiliza esos mecanismos

```
P(...) {  
    .  
    .  
    y = Q(x);  
    print(y)  
}  
  
int Q(int i) {  
    int t = 3 * i;  
    int v[10];  
    .  
    .  
    return v[0];  
}
```



# Soporte para procedimientos

- Transferencia del control
  - Necesario para “saltar” al código del procedimiento
  - Retorno al punto del salto
- Pasaje de datos
  - Argumentos
  - Valor de retorno
- Manejo de la memoria
  - Reservar lo necesario en el procedimiento
  - Liberar lo pedido al retornar
- Todos los mecanismos se implementan a través de instrucciones
- Todo procedimiento en x86-64 utiliza esos mecanismos

```
P(...) {  
    .  
    .  
    y = Q(x);  
    print(y)  
}
```

```
int Q(int i) {  
    int t = 3 * i;  
    int v[10];  
    .  
    .  
    return v[0];  
}
```

# Tabla de contenidos

---

## 1. Procedimientos

Mecanismos necesarios

Pila (*Stack*)

Calling conventions

Transferencia de control

Pasaje de datos

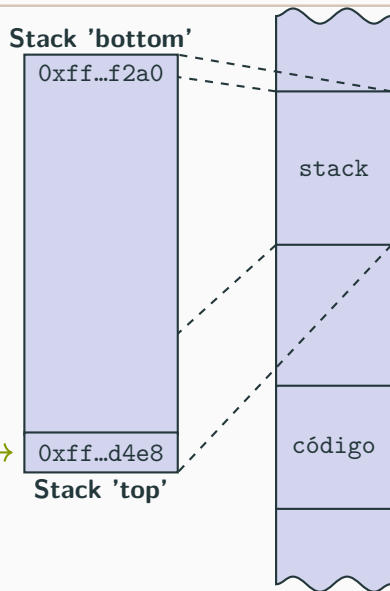
Manejo de datos locales

# 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

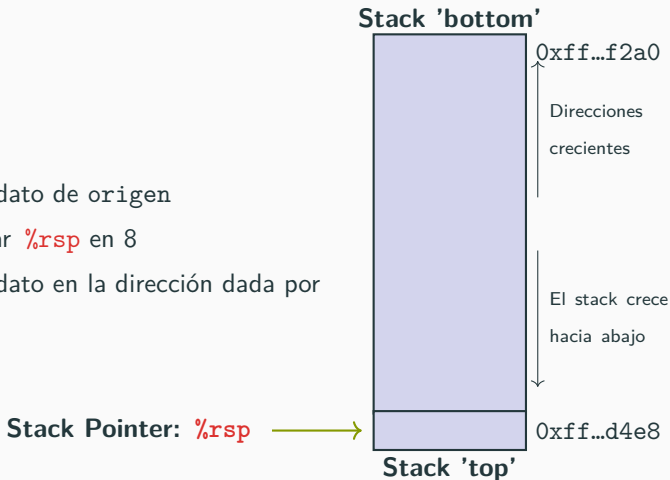
**Stack Pointer: %rsp** →



## Stack x86-64: push

`pushq origen`

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



## Stack x86-64: push

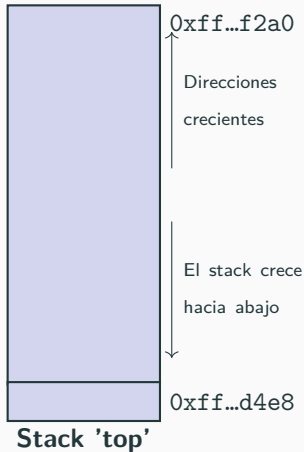
origen: 

`pushq origen`

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

Stack Pointer: `%rsp` 

Stack 'bottom'



## Stack x86-64: push

origen: valor

`pushq origen`

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

Stack Pointer: `%rsp`



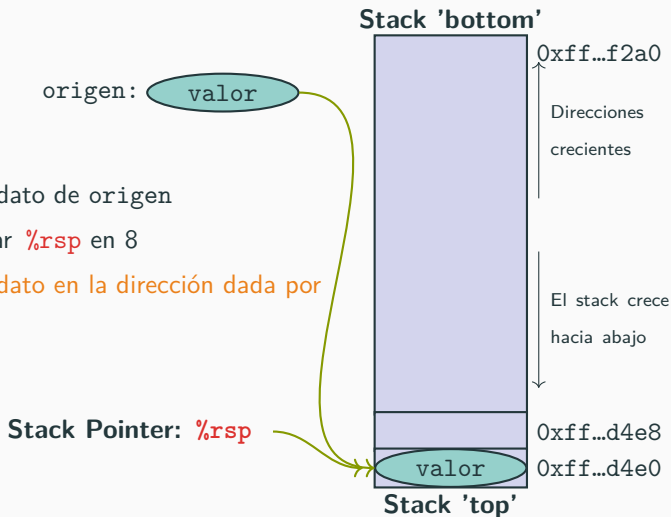
Stack 'bottom'



## Stack x86-64: push

`pushq origen`

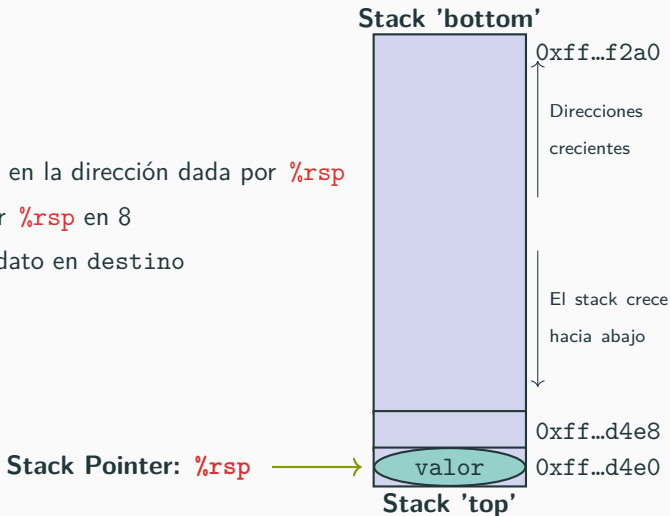
- obtener el dato de origen
- decrementar `%rsp` en 8
- guardar el dato en la dirección dada por `%rsp`



## Stack x86-64: pop

pop destino

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

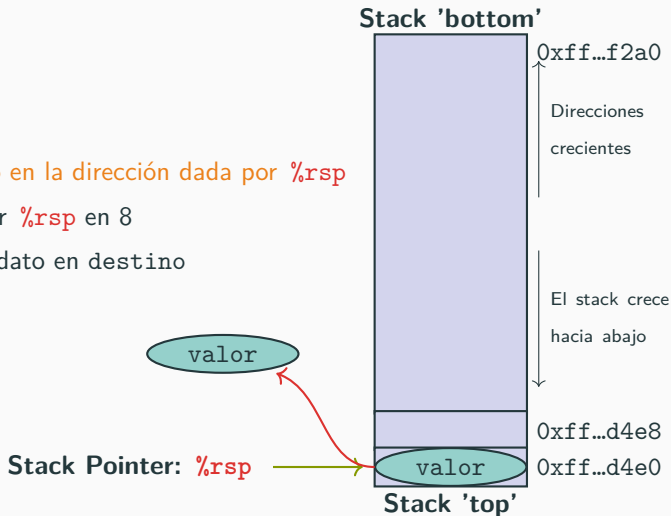




# Stack x86-64: pop

pop destino

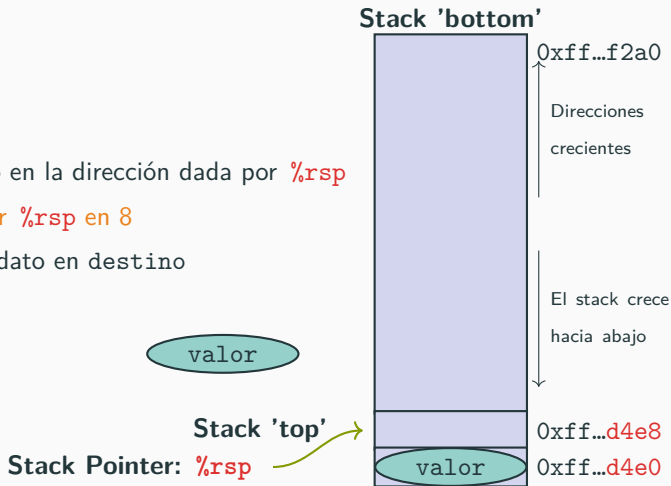
- leer el dato en la dirección dada por `%rsp`
- incrementar `%rsp` en 8
- guardar el dato en destino



# Stack x86-64: pop

pop destino

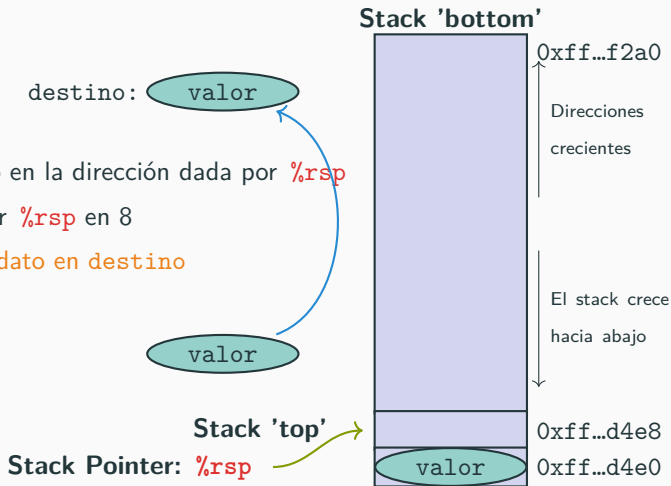
- leer el dato en la dirección dada por `%rsp`
- incrementar `%rsp` en 8
- guardar el dato en destino



## Stack x86-64: pop

pop destino

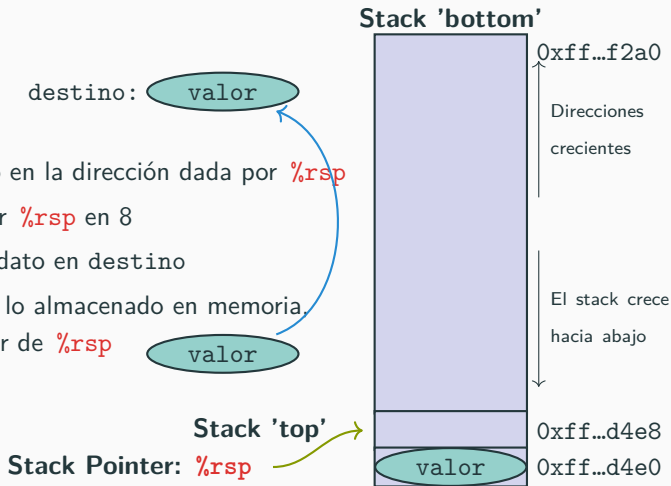
- leer el dato en la dirección dada por **%rsp**
- incrementar **%rsp** en 8
- guardar el dato en destino



## 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`



# Tabla de contenidos

---

## 1. Procedimientos

Mecanismos necesarios

Pila (*Stack*)

Calling conventions

Transferencia de control

Pasaje de datos

Manejo de datos locales

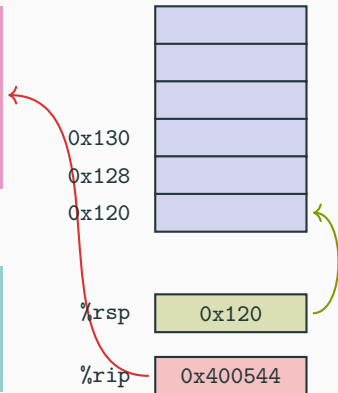
# Transferencia de control

- 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: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

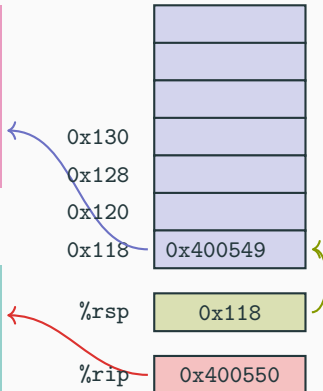
```
0000000000400550 <mult2>:  
400550: mov %rdi, %rax  
...  
400557: retq
```



## Transferencia de control: ejemplo

```
0000000000400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

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

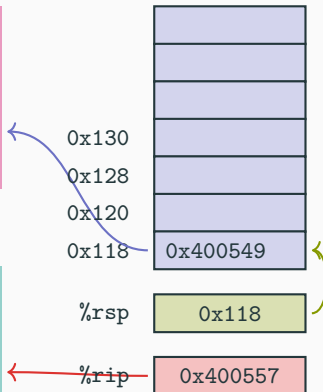




## Transferencia de control: ejemplo

```
0000000000400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

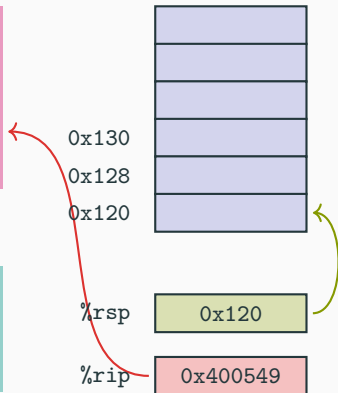
```
0000000000400550 <mult2>:  
400550: mov %rdi, %rax  
...  
400557: retq
```



## Transferencia de control: ejemplo

```
0000000000400540 <multstore>:  
...  
...  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
...  
...
```

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



# Pasaje de datos

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

- Primeros 6 argumentos**
- Argumentos siguientes**

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

Stack bottom

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

Stack top

- Valor de retorno**

%rax
------

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

## Pasaje de datos: ejemplo

```
void multstore
(long x, long y, long *
 dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
# x en %rdi, y en %rsi, dest en %rdx
...
400541: mov        %rdx, %rbx # guarda dest
400544: callq     400550 <mult2>
# t en %rax
400549: mov        %rax, (%rbx)
```

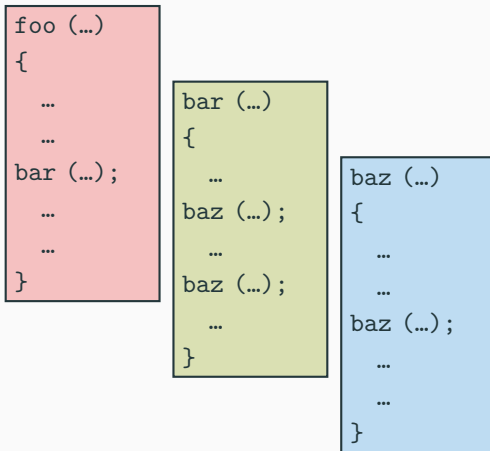
```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
# a en %rdi, b en %rsi
400550: mov        %rdi,%rax
400553: imul       %rsi,%rax
# s en %rax
400557: retq
```

# Lenguajes basados en pilas

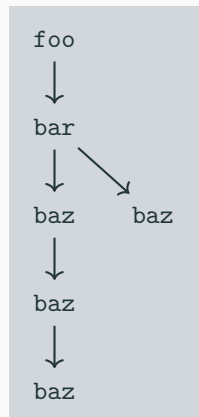
- Lenguajes que soportan recursividad
  - El código debe ser reentrante (*reentrant*)
    - Para soportar instanciaciones múltiples de un mismo procedimiento
  - Requiere de espacio para almacenar el estado de cada instancia
    - Argumentos
    - Variables locales
    - Retorno
- Disciplina del stack
  - Necesita el estado de un procedimiento durante un tiempo finito
    - Desde que se lo llama hasta que termina
  - El proceso invocado finaliza **antes** que el invocante
- El stack se reserva de a **frames**
  - Guarda el estado de una única instancia de un procedimiento

## Ejemplo de cadena de invocaciones



baz() es recursivo

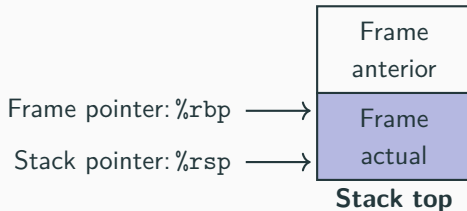
### Árbol de llamadas



# Stack frames

## ■ Contiene:

- Información de retorno
- Almacenamiento local
- Espacio temporal

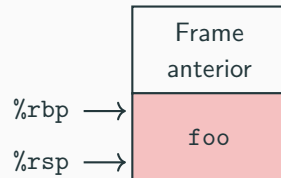


## ■ Administración:

- 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`

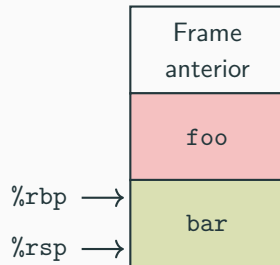
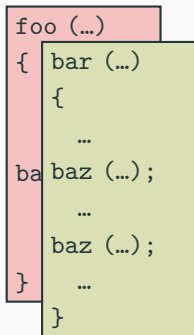
## Stack frames: ejemplo

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

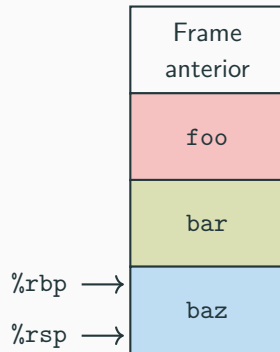
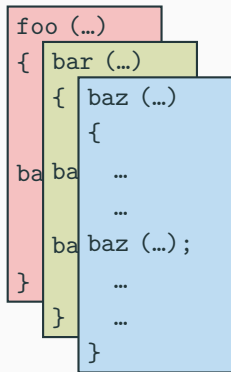




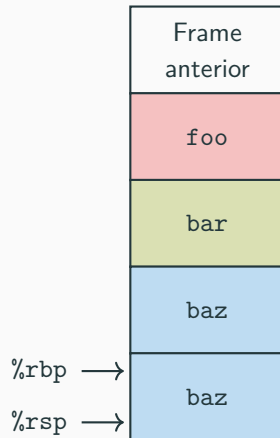
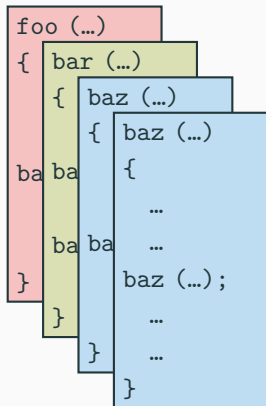
## Stack frames: ejemplo



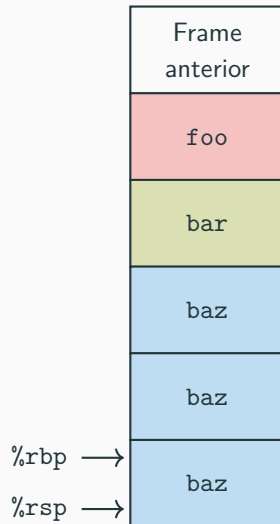
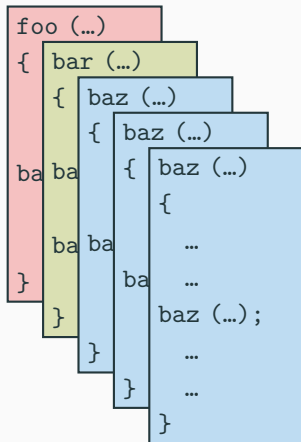
## Stack frames: ejemplo



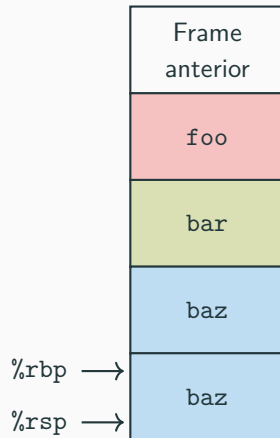
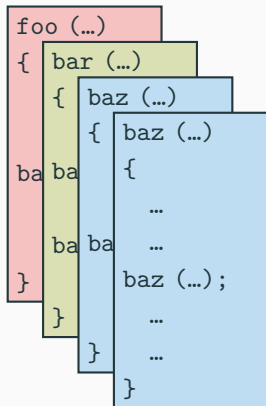
## Stack frames: ejemplo



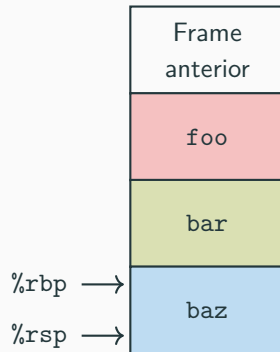
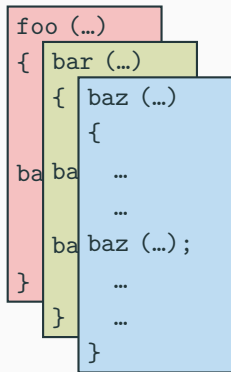
## Stack frames: ejemplo



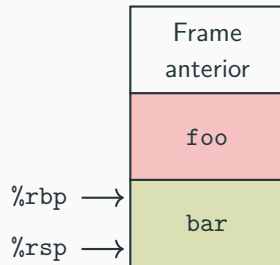
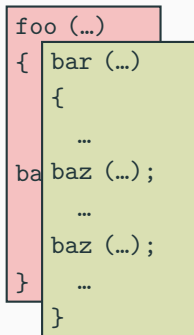
# Stack frames: ejemplo



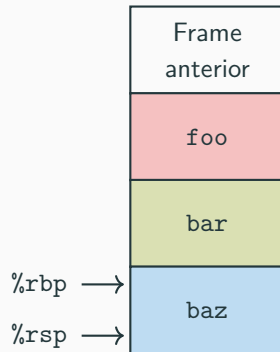
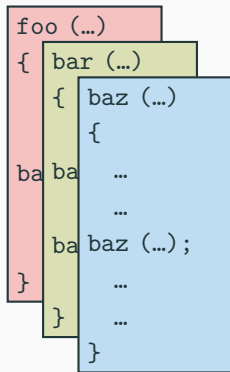
## Stack frames: ejemplo



## Stack frames: ejemplo

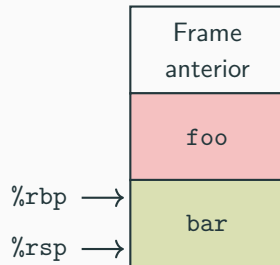
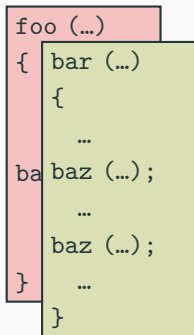


## Stack frames: ejemplo



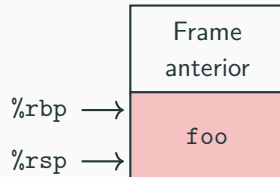


## Stack frames: ejemplo



## Stack frames: ejemplo

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



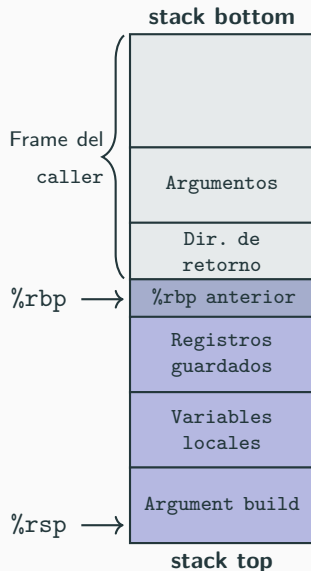
# Linux Stack Frame

## ■ 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 `callq`
- argumentos para esta función
  - En x86\_64 (64 bits): del séptimo en adelante
  - En x86 (32 bits): todos



# 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?

```
foo:
    ...
    movq    $b00710ad, %rdx
    callq   bar
    addq    %rdx, %rax
    ...
    ret
```

```
bar:
    ...
    subq    $deadbeef, %rdx
    ...
    ret
```

- El contenido de **%rdx** es sobrescrito por bar
- Es necesario algún arreglo de partes para que funcione correctamente

# 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 sobrescrito 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

Valor de retorno

%rax

Argumentos

%rdi

%rsi

%rdx

%rsx

%r8

%r9

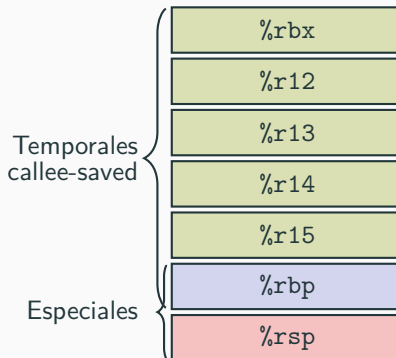
Temporales  
caller-saved

%r10

%r11

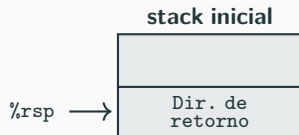
# Convenciones para registros **callee-saved** en x86\_64

- **%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



## Ejemplo de *calling conventions*

```
long f1(long x) {  
    long n = 481516;  
    long y = f2(&n, 2342)  
    return x + y  
}
```

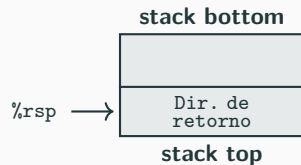


- x se pasa en %rdi
- %rdi se necesita para llamar a fun()
- x (%rdi) se necesita *después* de llamar a fun(), *che facciamo?*



## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```



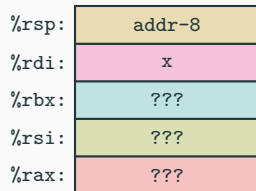
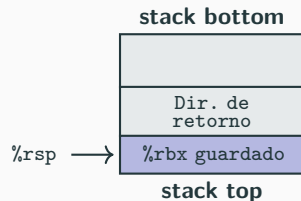
```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

%rsp:	addr
%rdi:	x
%rbx:	???
%rsi:	???
%rax:	???

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

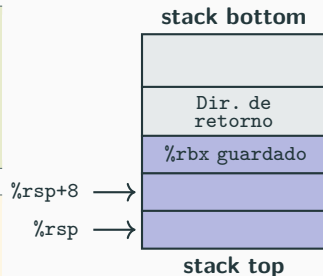
```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

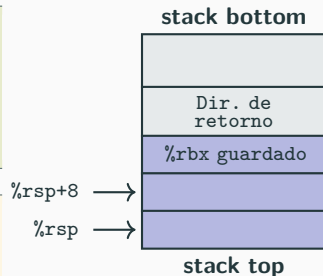


%rsp:	addr-24
%rdi:	x
%rbx:	???
%rsi:	???
%rax:	???

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

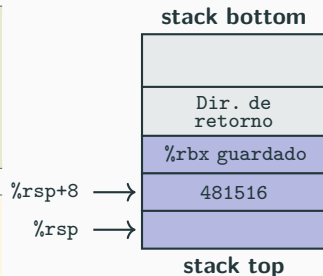


%rsp:	addr-24
%rdi:	x
%rbx:	x
%rsi:	???
%rax:	???

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

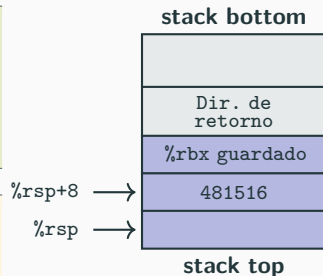


%rsp:	addr-24
%rdi:	x
%rbx:	x
%rsi:	???
%rax:	???

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

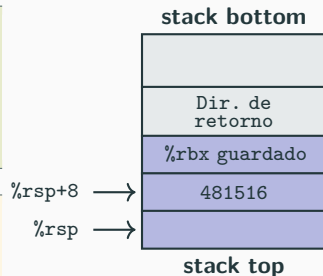


%rsp:	addr-24
%rdi:	x
%rbx:	x
%rsi:	2342
%rax:	???

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

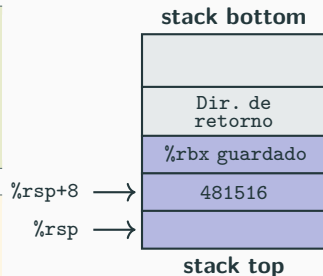


%rsp:	addr-24
%rdi:	%rsp + 8 = &n
%rbx:	x
%rsi:	2342
%rax:	???

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



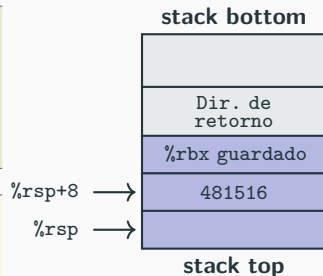
%rsp:	addr-24
%rdi:	addr-16
%rbx:	x
%rsi:	2342
%rax:	???



## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



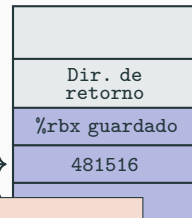
%rsp:	addr-24
%rdi:	addr-16
%rbx:	x
%rsi:	2342
%rax:	???

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

stack bottom



%rsp+8 →

Se ejecuta la llamada a f2():

- x está almacenado en %rbx

Al retornar:

- y está en %rax

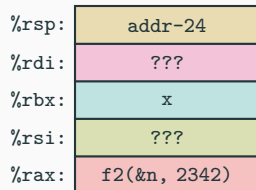
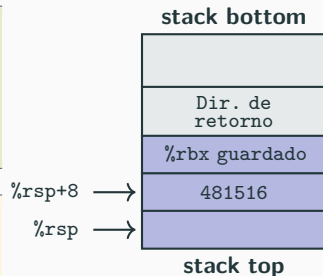
Por cómo funciona el stack, “nada” más cambió.

%rax: f2(&n, 2342)

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

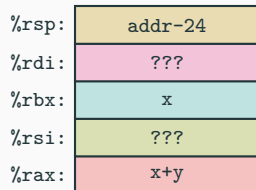
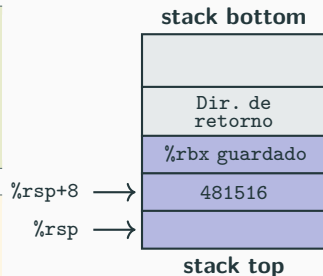
```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

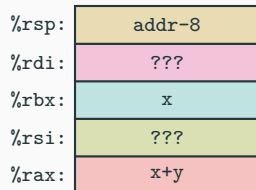
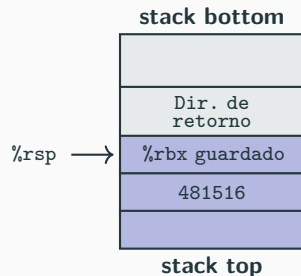
```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

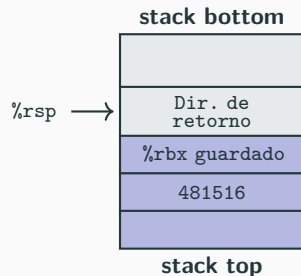
```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```

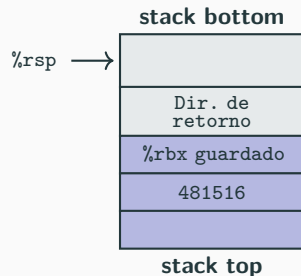


%rsp:	addr
%rdi:	???
%rbx:	???
%rsi:	???
%rax:	x+y

## Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
    pushq %rbx
    subq $16, %rsp
    movq %rdi, %rbx
    movq $481516, 8(%rsp)
    movl $2342, %esi
    leaq 8(%rsp), %rdi
    call f2
    addq %rbx, %rax
    addq $16, %rsp
    popq %rbx
    ret
```



%rsp:	addr+8
%rdi:	???
%rbx:	???
%rsi:	???
%rax:	x+y

# Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`

