

Programación a Nivel-Máquina III: Procedimientos

Estructura de Computadores
Semana 5

Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIT/C.1 BRY com

Transparencias del libro CS:APP, Cap.3

Introduction to Computer Systems: a Programmer's Perspective

Autores: Randal E. Bryant y David R. O'Hallaron

<http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html>

Guía de trabajo autónomo (4h/s)

■ **Lectura:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Procedures
 - § 3.7 pp.274-291
- Understanding Pointers, Using GDB.
 - § 3.10.1-2 pp.312-316

■ **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.32 § 3.7.2, p.280
- Probl. 3.33 § 3.7.3, p.282
- Probl. 3.34 § 3.7.5, p.288
- Probl. 3.35 § 3.7.6, p.290

Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIIT/[C.1 BRY com](#)

Programación Máquina III: Procedimientos

- **Procedimientos**
 - Mecanismos
 - Estructura de la pila
 - Convenciones de llamada
 - Pasando el control
 - Pasando los datos
 - Gestionando datos locales
 - Ejemplos ilustrativos de Recursividad

Mecanismos en los Procedimientos

■ Transferencia de control

- al principio del código del procedimiento
- de vuelta al punto de retorno

■ Transferencia de datos

- Argumentos del procedimiento
- Valor de retorno

■ Gestión de memoria

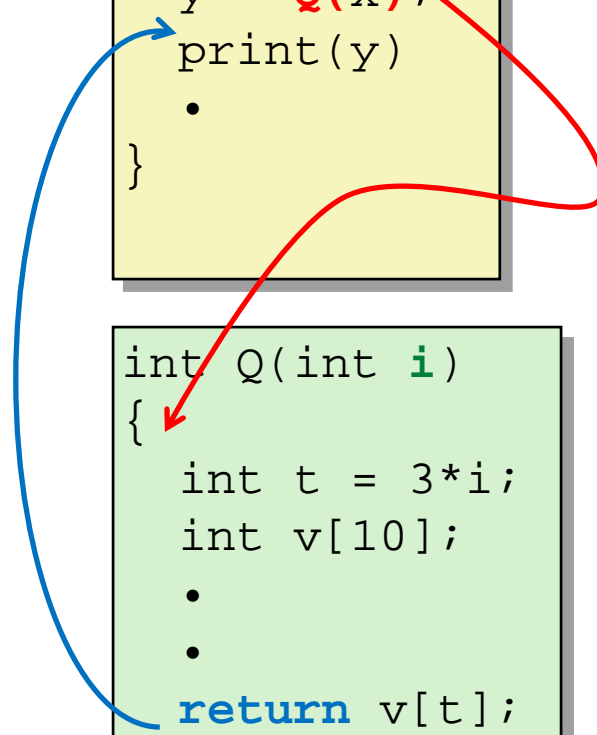
- Reservar[†] durante ejecución procedimiento
- Liberar al retornar

■ Mecanismos todos implementados con instrucciones máquina

- La implement. x86-64 de un proc. concreto usa sólo los mecanismos que éste requiera

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

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



Mecanismos en los Procedimientos

■ Transferencia de control

- al principio del código del procedimiento
- de vuelta al punto de retorno

■ Transferencia de datos

- Argumentos del procedimiento
- Valor de retorno

■ Gestión de memoria

- Reservar[†] durante ejecución procedimiento
- Liberar al retornar

■ Mecanismos todos implementados con instrucciones máquina

■ La implement. x86-64 de un proc. concreto usa sólo los mecanismos que éste requiera

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

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

Mecanismos en los Procedimientos

■ Transferencia de control

- al principio del código del procedimiento
- de vuelta al punto de retorno

■ Transferencia de datos

- Argumentos del procedimiento
- Valor de retorno

■ Gestión de memoria

- Reservar[†] durante ejecución procedimiento
- Liberar al retornar

■ Mecanismos todos implementados con instrucciones máquina

- La implement. x86-64 de un proc. concreto usa sólo los mecanismos que éste requiera

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

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

[†] "allocate" = ubicar 6

Mecanismos en los Procedimientos

■ Transferencia de control

- al principio del código del procedimiento
- de vuelta al punto de retorno

■ Transferencia de datos

- Argumentos del procedimiento
- Valor de retorno

■ Gestión de memoria

- Reservar[†] durante ejecución procedimiento
- Liberar al retornar

■ Mecanismos todos implementados con instrucciones máquina

■ La implement. x86-64 de un proc. concreto usa sólo los mecanismos que éste requiera

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

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

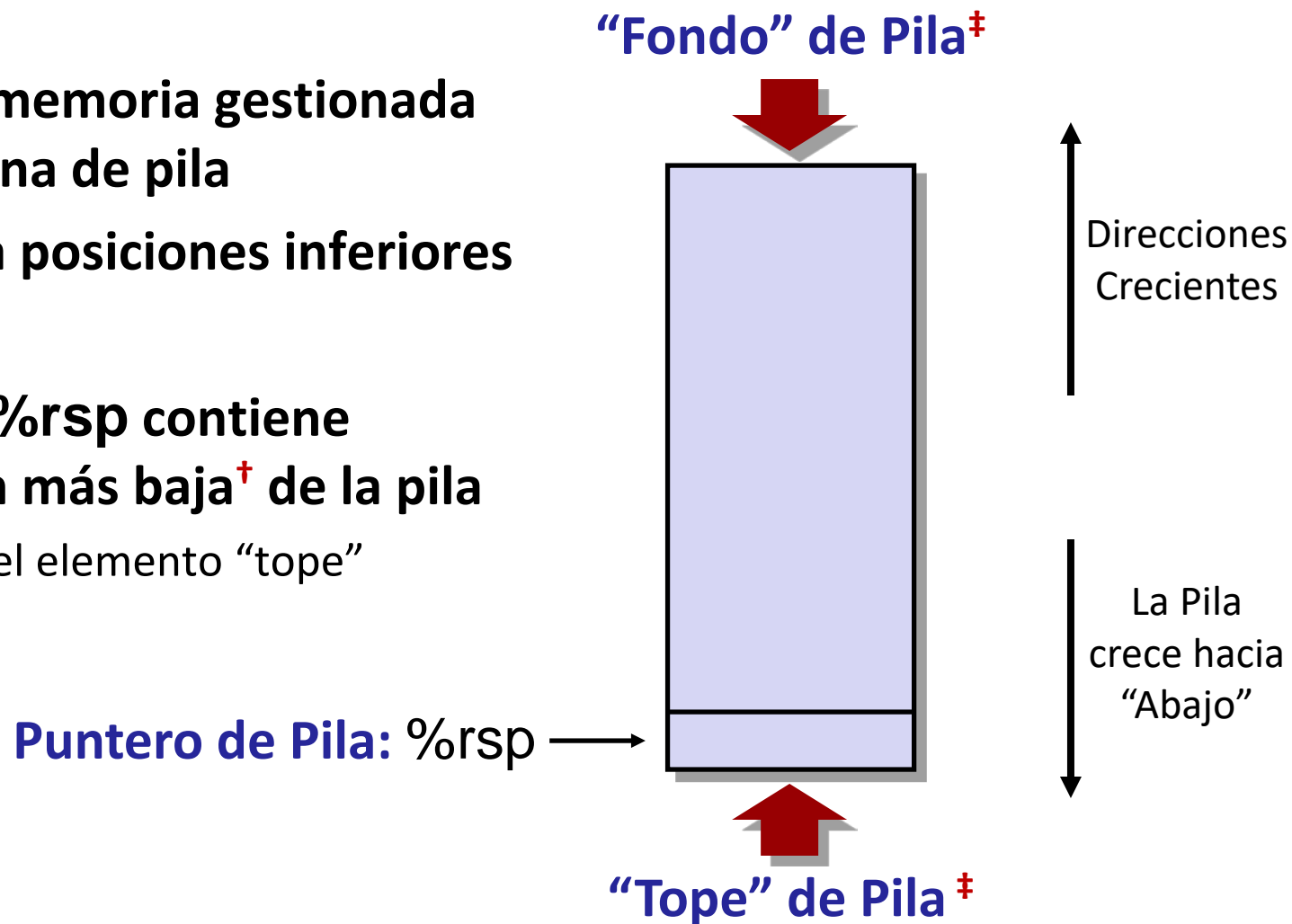
[†] "allocate" = ubicar 7

Programación Máquina III: Procedimientos

- **Procedimientos**
 - Mecanismos
 - Estructura de la pila
 - Convenciones de llamada
 - Pasando el control
 - Pasando los datos
 - Gestionando datos locales
 - Ejemplos ilustrativos de Recursividad

Pila x86-64

- Región de memoria gestionada con disciplina de pila
- Crece hacia posiciones inferiores
- El registro `%rsp` contiene la dirección más baja[†] de la pila
 - dirección del elemento “tope”



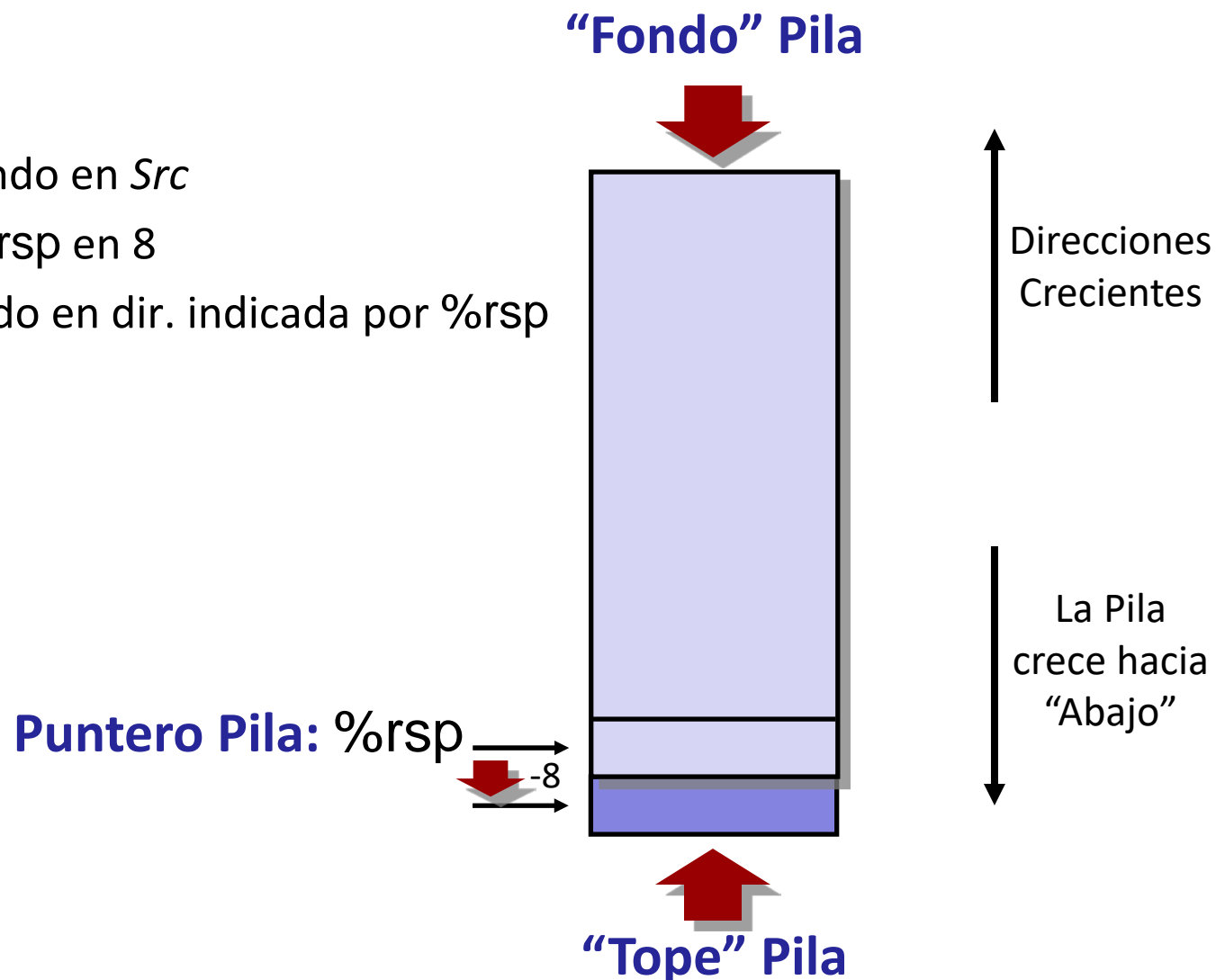
[†] “lowest” en el instante actual

[‡] “top” = tope, “bottom” = fondo 9

Pila x86-64: Push[†]

■ `pushq Src`

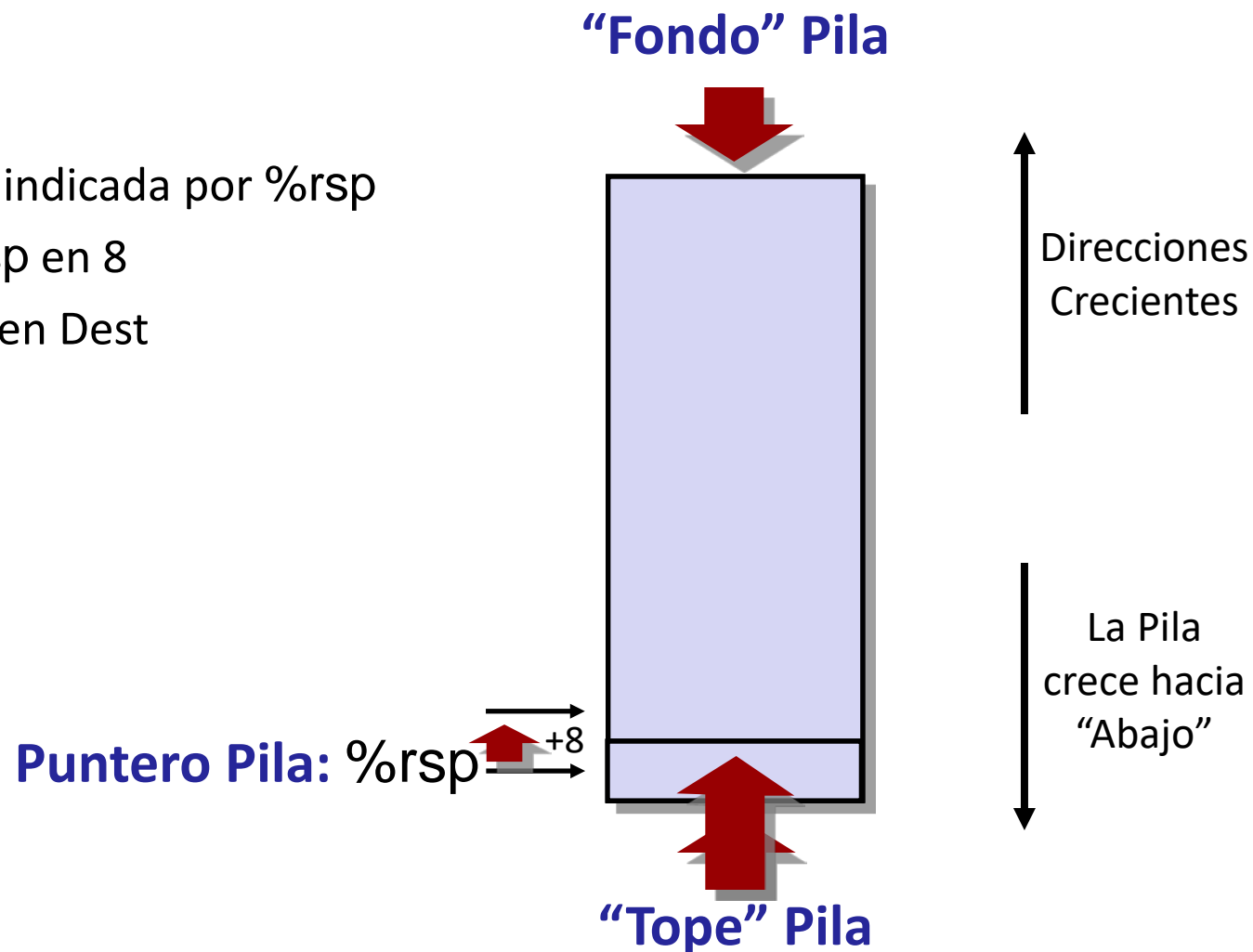
- Capta el operando en *Src*
- Decrementa `%rsp` en 8
- Escribe operando en dir. indicada por `%rsp`



Pila x86-64: Pop[†]

■ `popq Dest`

- Lee valor de dir. indicada por `%rsp`
- Incrementa `%rsp` en 8
- Almacena valor en `Dest`



Programación Máquina III: Procedimientos

- **Procedimientos**
 - Mecanismos
 - Estructura de la pila
 - Convenciones de llamada
 - **Pasando el control**
 - Pasando los datos
 - Gestionando datos locales
 - Ejemplos ilustrativos de Recursividad

Código ejemplo

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

```
0000000000400540 <multstore>:
  400540: push    %rbx                # preservar %rbx
  400541: mov     %rdx,%rbx           # conservar dest
  400544: callq   400550 <mult2>      # mult2(x,y)
  400549: mov     %rax, (%rbx)         # salvar en dest
  40054c: pop     %rbx                # restaurar %rbx
  40054d: retq                      # retornar
```

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

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

Flujo de Control en Procedimientos

- Usar la pila para soportar llamadas y retornos de procedimientos
- **Llamada a procedimiento: `call label`**
 - Recuerda[†] la dirección de retorno en la pila
 - Salta a etiqueta ***label***
 - Codificada con *direccionamiento relativo a IP*
- **Dirección de retorno:**
 - Dirección de la siguiente instrucción justo después de la llamada (`call`)
 - Ejemplo en el desensamblado anterior: **0x400549**
- **Retorno de procedimiento: `ret`**
 - Recupera[†] la dirección (de retorno) de la pila
 - Salta a dicha dirección

Ejemplo Flujo Control #1

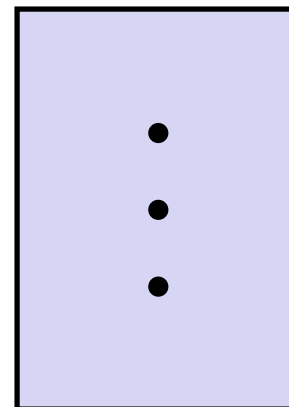
```
0000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov    %rax, (%rbx)
.
.
```

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

0x130

0x128

0x120



%rsp

0x120

%rip

0x400544

■ Direcccionamiento relativo a contador de programa (RIP)

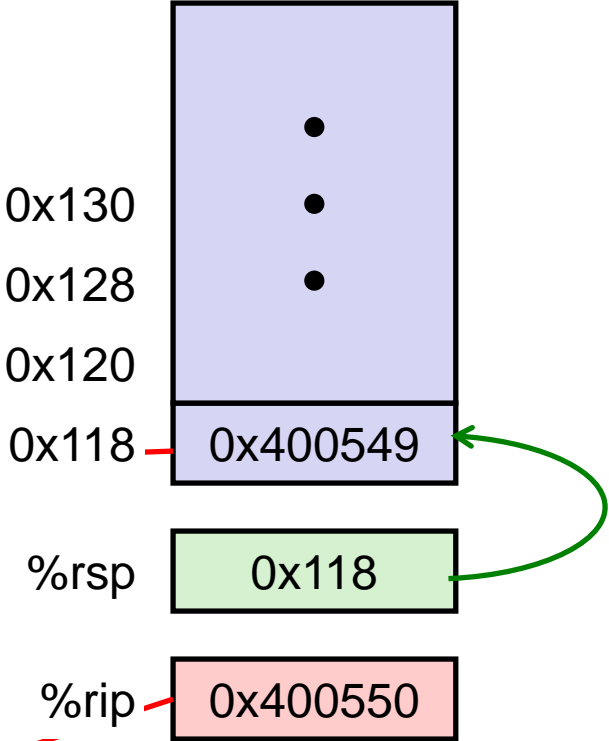
RIP	0x00400549	(tras fetch)
+offs	0x00000007	
=Dst	0x00400550	

Ejemplo Flujo Control #2

```
00000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov    %rax, (%rbx)
.
.
```

```
00000000000400550 <mult2>:
400550: mov    %rdi,%rax
.
.
400557: retq
```

```
400544: e8 07 00 00 00    callq 400550 <mult2>
400549: 48 89 03          mov    %rax, (%rbx)
40054c: 5b               pop    %rbx
40054d: c3              retq
40054e: 66 90           nop
00000000000400550 <mult2>:
400550: 48 89 f8          mov    %rdi,%rax
```



Ejemplo Flujo Control #3

```
0000000000400540 <multstore>:
```

•
•
•

```
400544: callq 400550 <mult2>
```

```
400549: mov %rax, (%rbx) ←
```

•
•

```
0000000000400550 <mult2>:
```

```
400550: mov %rdi, %rax
```

•
•

```
400557: retq ←
```

0x130

0x128

0x120

0x118

0x400549

%rsp

0x118

%rip

0x400557

Ejemplo Flujo Control #4

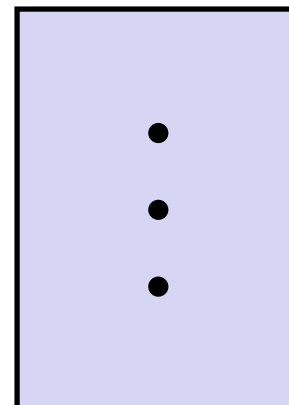
```
0000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov   %rax, (%rbx)
.
.
```

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

0x130

0x128

0x120

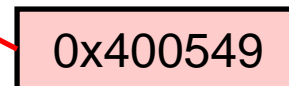
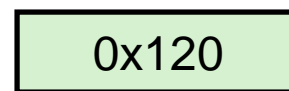


%rsp

0x120

%rip

0x400549



Programación Máquina III: Procedimientos

- **Procedimientos**
 - Mecanismos
 - Estructura de la pila
 - Convenciones de llamada
 - Pasando el control
 - **Pasando los datos**
 - Gestionando datos locales
 - Ejemplos ilustrativos de Recursividad

Flujo de Datos para Procedimientos

Registros

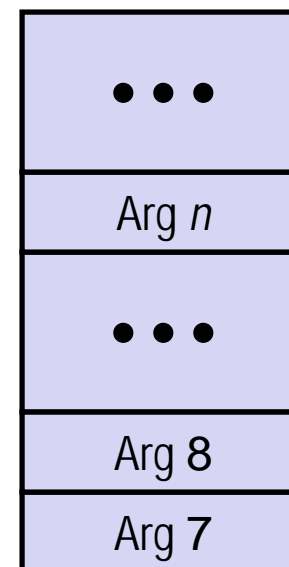
■ Primeros 6 argumentos

%rdi
%rsi
%rdx
%rcx
%r8
%r9

■ Valor de retorno

%rax

Pila



- Sólo se reserva espacio en la pila cuando se necesita

Ejemplo

Flujo Datos

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

```
00000000000400540 <multstore>:
    # x en %rdi, y en %rsi, dest en %rdx
400540: push    %rbx                # preservar %rbx
400541: mov     %rdx,%rbx           # conservar dest
400544: callq   400550 <mult2>      # mult2(x,y)
400549: mov     %rax,(%rbx)         # salvar en dest
40054c: pop     %rbx                # restaurar %rbx
    # %rax libre (void), todavía conserva t=x*y
40054d: retq                      # retornar
```

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

```
00000000000400550 <mult2>:
    # a en %rdi, b en %rsi
400550: mov     %rdi,%rax           # a
400553: imul    %rsi,%rax           # a * b
    # s en %rax
400557: retq                      # retornar
```

Programación Máquina III: Procedimientos

- **Procedimientos**
 - Mecanismos
 - Estructura de la pila
 - Convenciones de llamada
 - Pasando el control
 - Pasando los datos
 - **Gestionando datos locales**
 - Ejemplos ilustrativos de Recursividad

Lenguajes basados en pila[†]

■ Lenguajes que soportan recursividad

- P.ej., C, Pascal, Java
- El código debe ser “*Reentrante*”
 - Múltiples instanciaciones[‡] simultáneas de un mismo procedimiento
- Se necesita algún lugar para guardar el estado de cada instancia
 - Argumentos
 - Variables locales
 - Puntero (dirección) de retorno

■ Disciplina de pila

- Estado para un procedimiento dado, necesario por tiempo limitado
 - Desde que se le llama hasta que retorna
- El invocado[‡] retorna antes de que lo haga el invocante[‡]

■ La pila se reserva en *Marcos*[‡]

- estado para una sola instancia de procedimiento

[†] “block structured” en terminología Intel

[‡] “callee/caller” en inglés

[‡] “allocated in frames”

[‡] “instantiate” = crear nuevos ejemplares 23

Ejemplo de secuencia[†] de llamadas

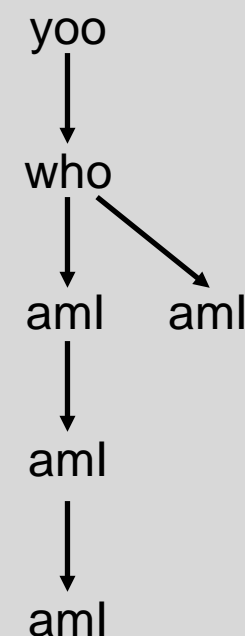
```
yoo (...)  
{  
  .  
  .  
  who ( ) ;  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI ( ) ;  
  . . .  
  amI ( ) ;  
  . . .  
}
```

```
amI (...)  
{  
  .  
  .  
  amI ( ) ;  
  .  
  .  
}
```

El procedimiento amI() es recursivo

Ejemplo Sec. Llamadas



[†] "call chain".
 "yoo/you" = tú,
 "who" = quién,
 "am I" = soy yo. 24

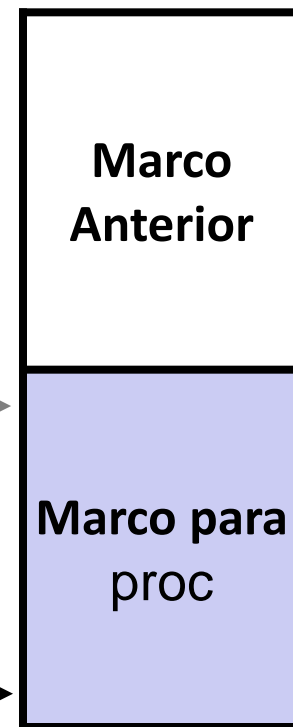
Marcos de Pila

■ Contenido

- Información de retorno
- Almacnmto[†] local (si necesario)
- Espacio temporal (si necesario)

*Puntero de Marco: %rbp[‡]
(Opcional)*

Puntero de Pila: %rsp





“Tope” Pila

■ Gestión

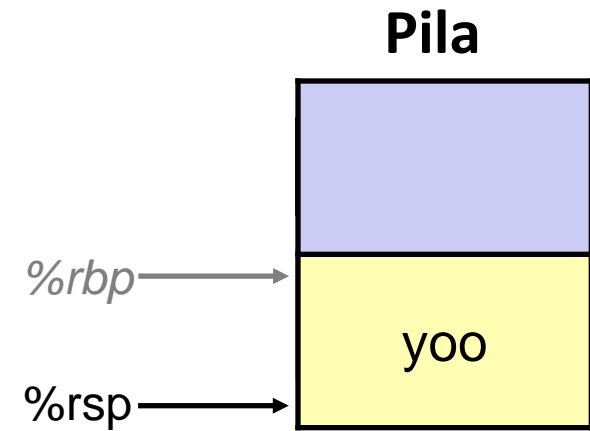
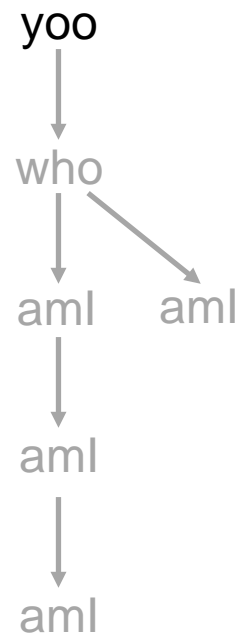
- Espacio se reserva al entrar el procedimiento
 - Código de “Inicialización”[‡]
 - Incluye el “push dir.ret.” de la instrucción **call**
- Se libera al retornar
 - Código de “Finalización”[‡]
 - Incluye el “pop cont.prog.” de la instrucción **ret**

[‡] si se usa `-fno-omit-frame-pointer`
[‡] “set-up/finish code”
[†] “local storage” 25

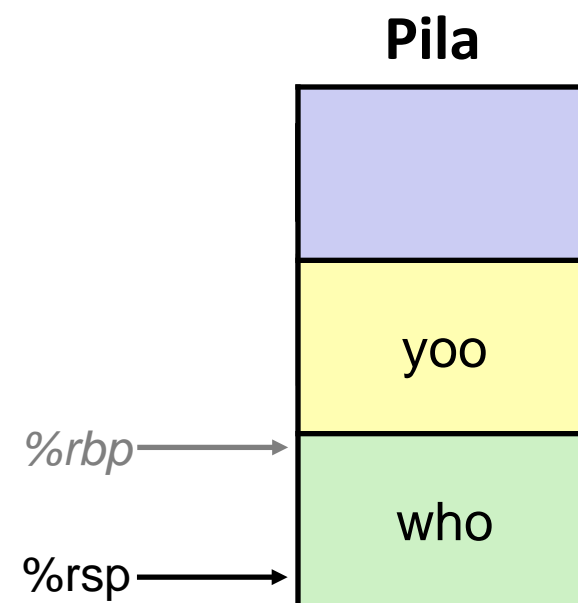
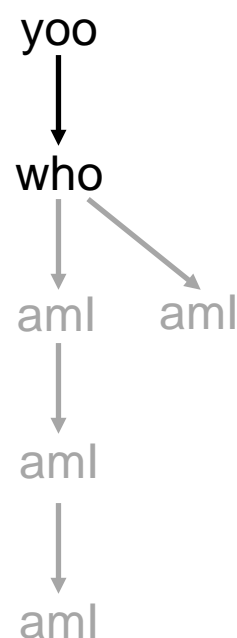
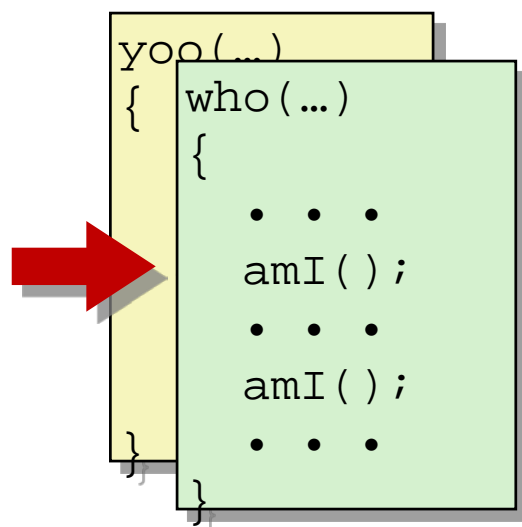
Ejemplo



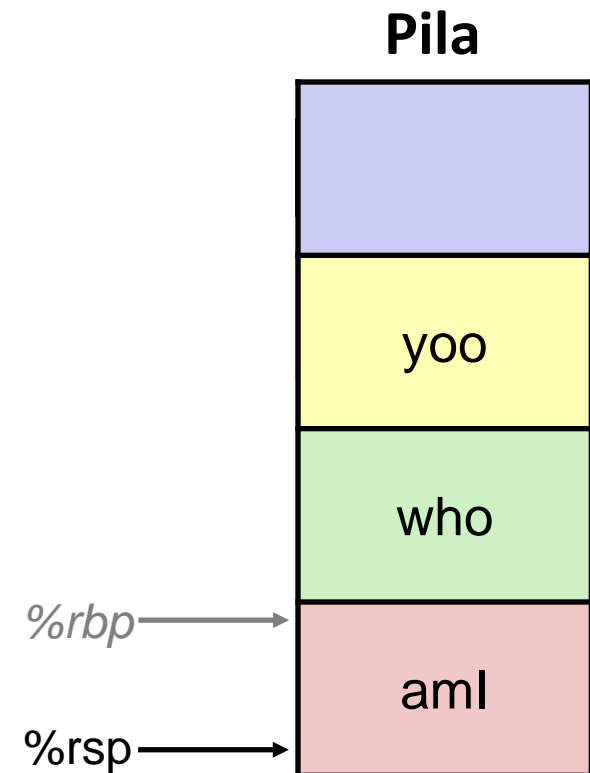
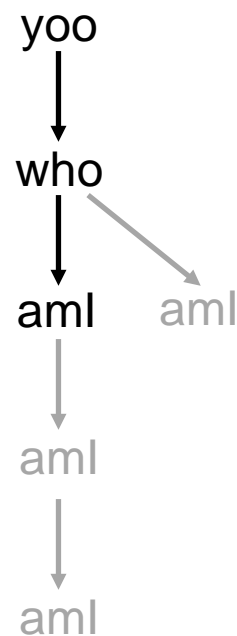
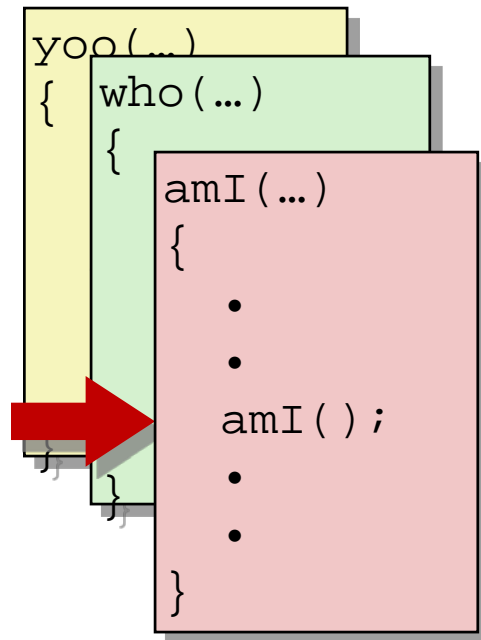
```
yoo ( ... )
{
    .
    .
    who ( ) ;
    .
    .
}
```



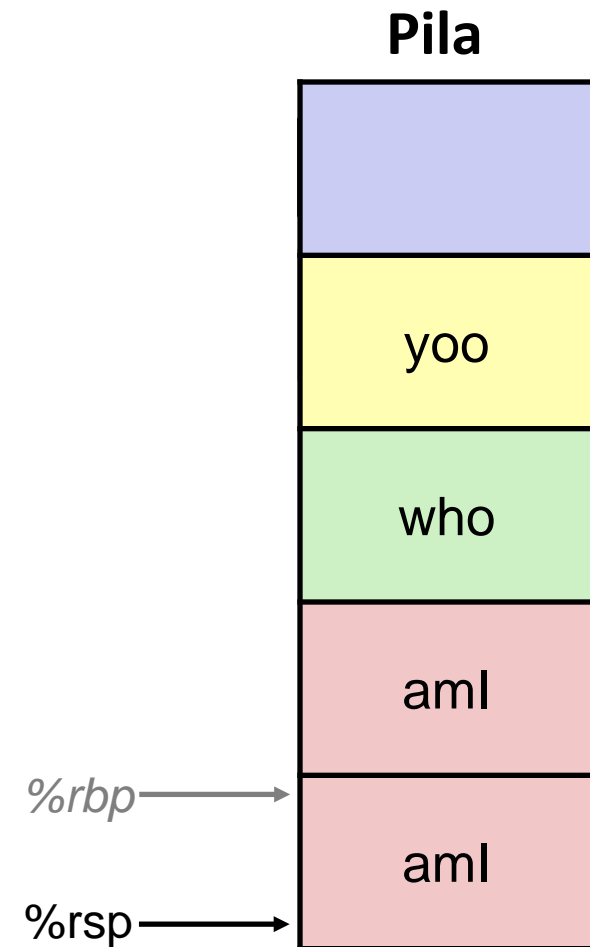
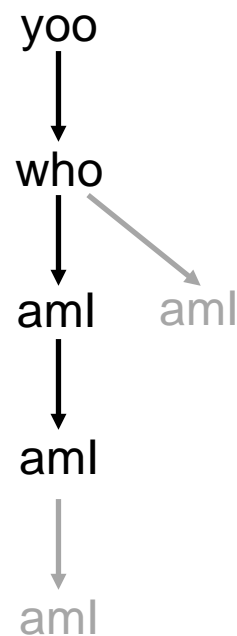
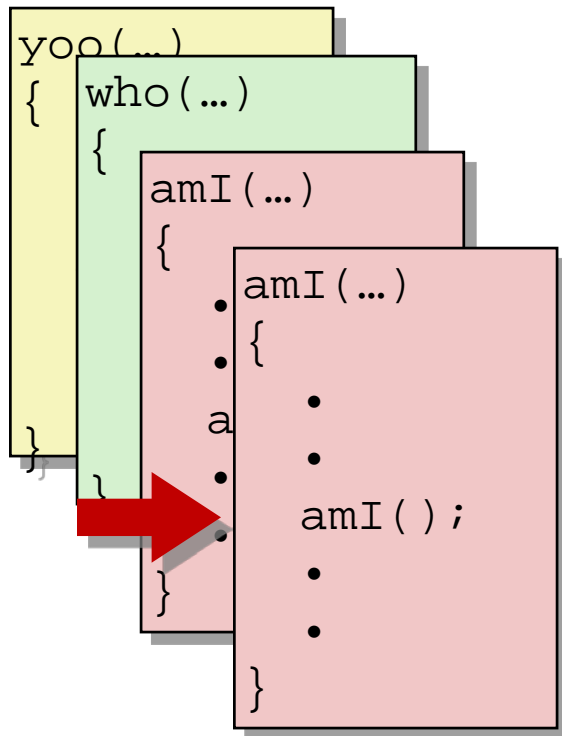
Ejemplo



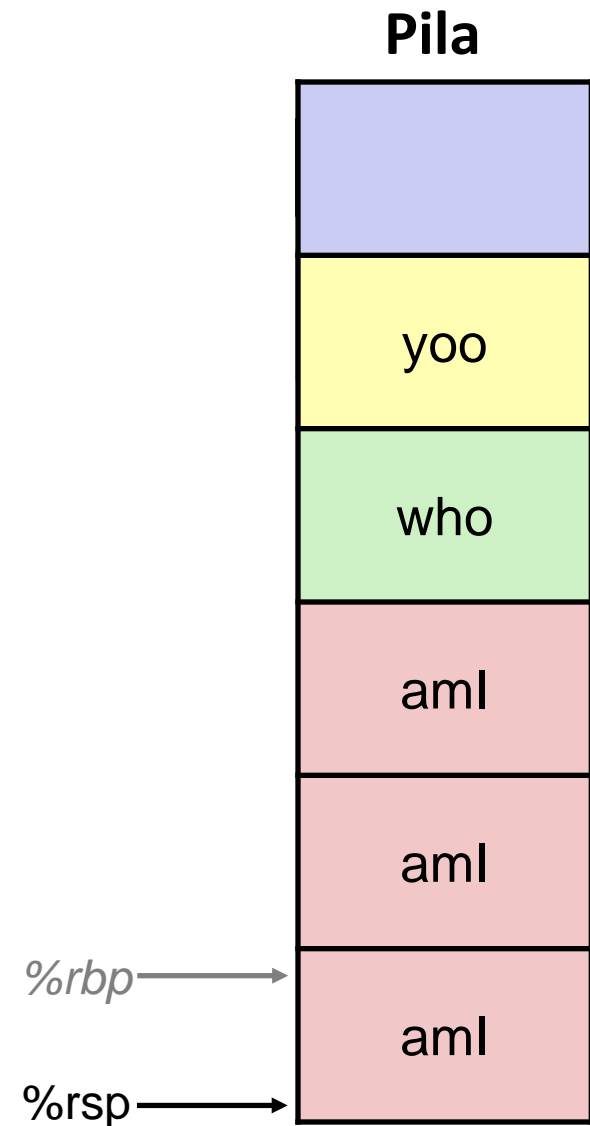
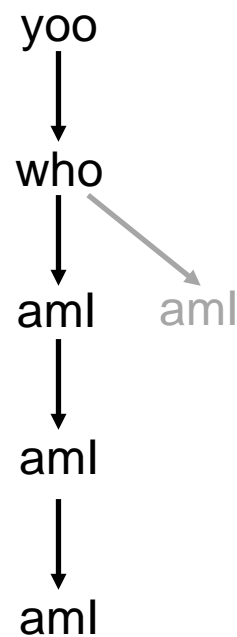
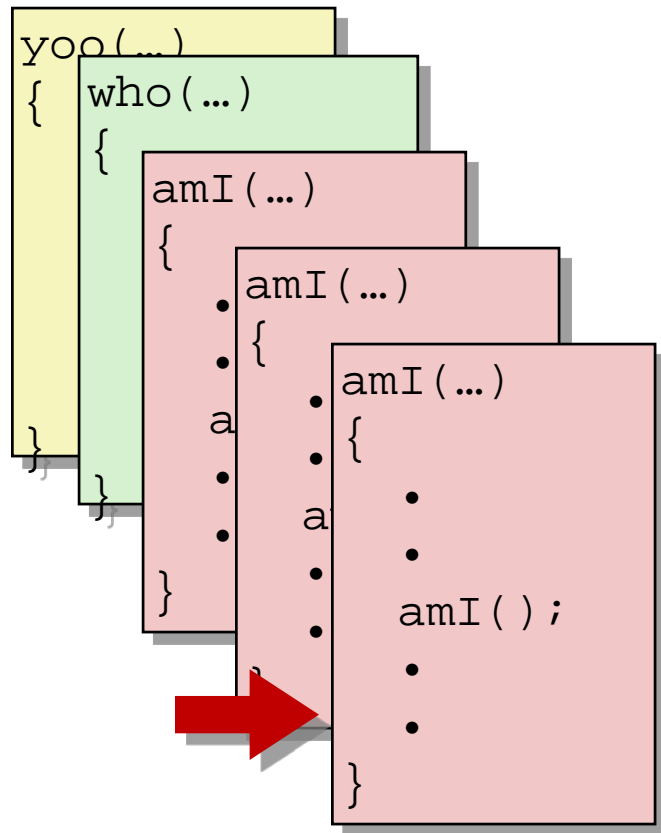
Ejemplo



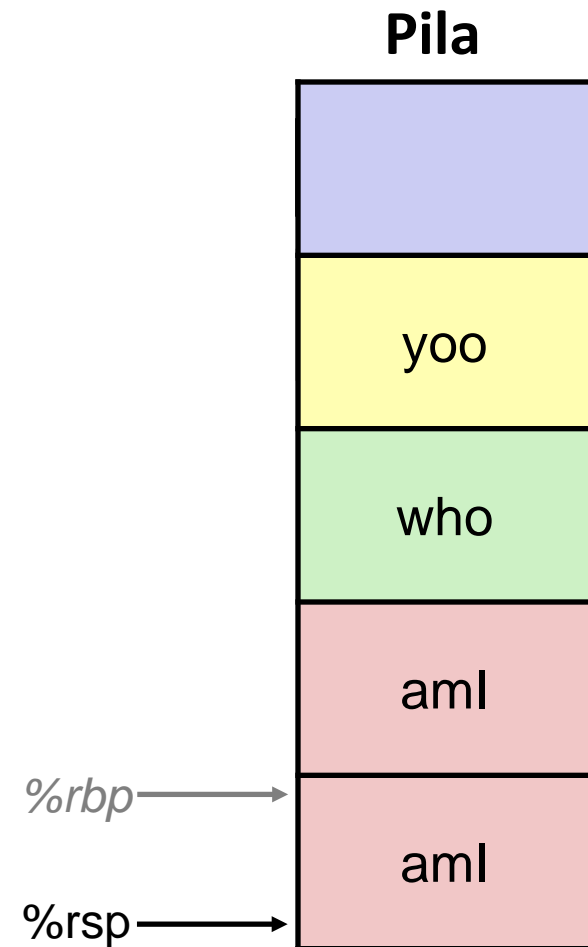
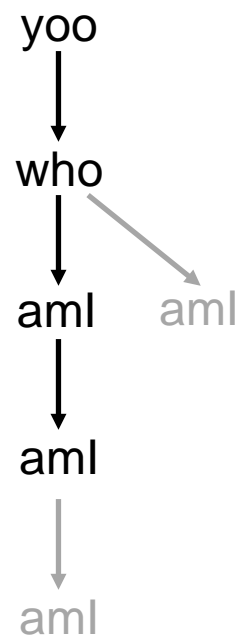
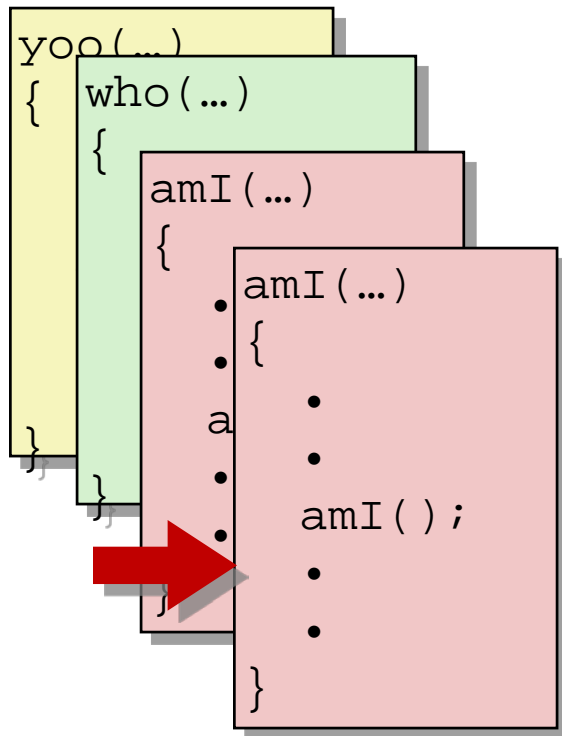
Ejemplo



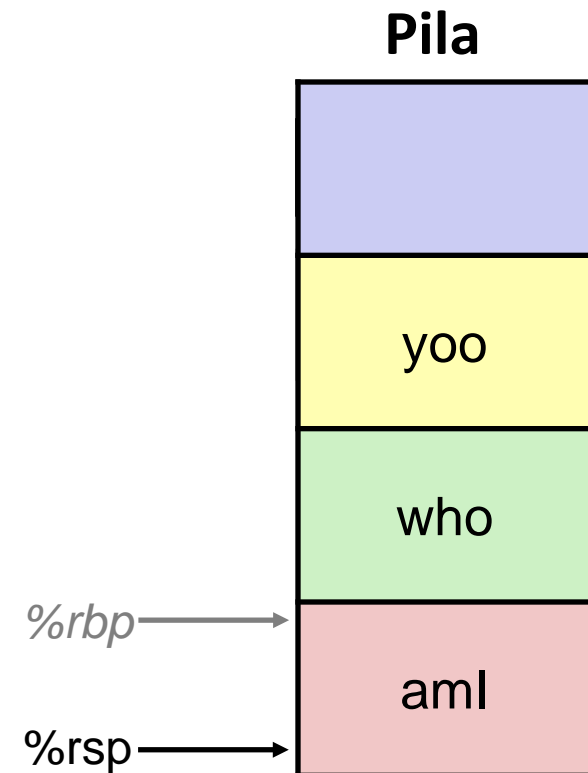
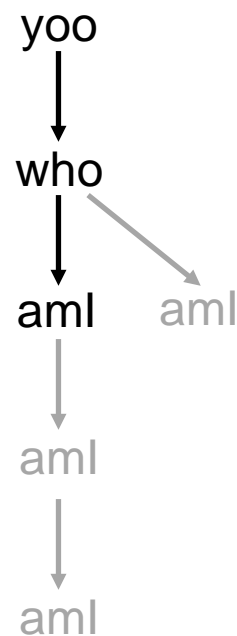
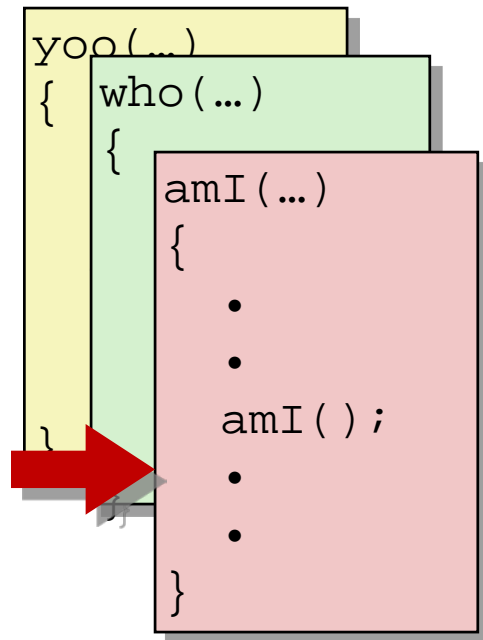
Ejemplo



Ejemplo



Ejemplo

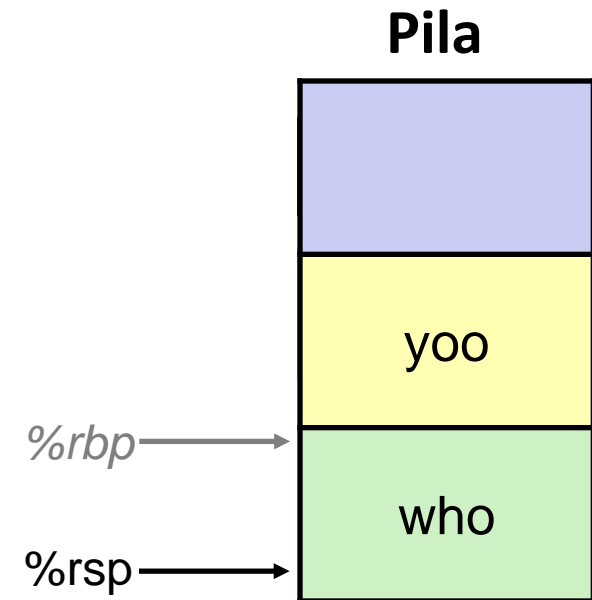
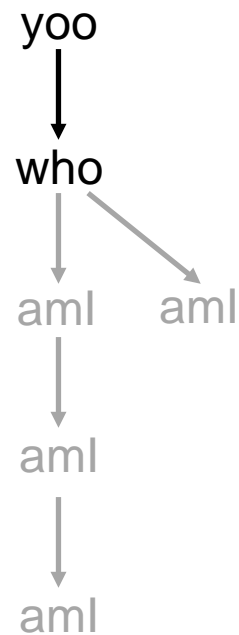



```

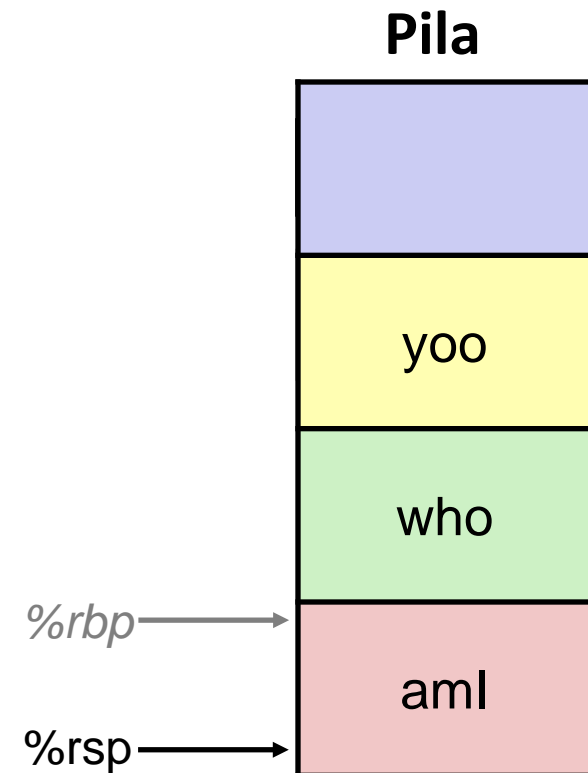
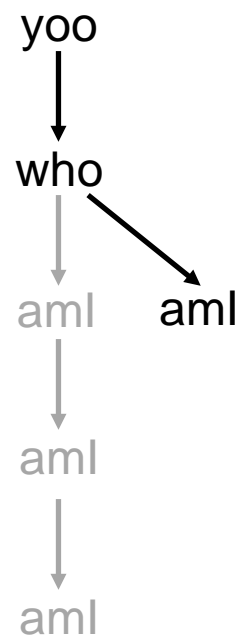
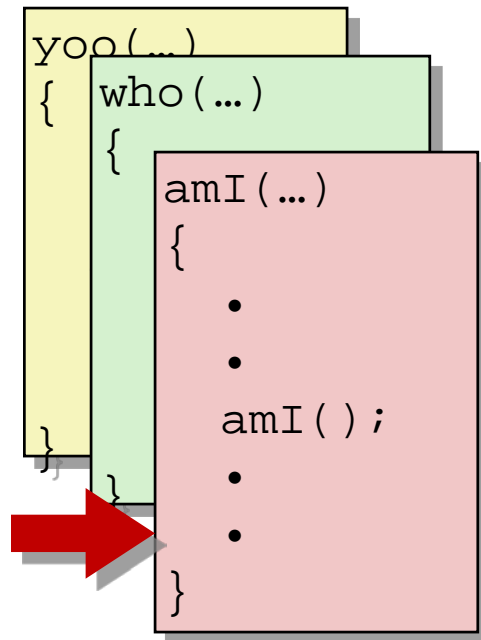
yoo(...)
{
  who(...)
  {
    • • •
    amI ( ) ;
    • • •
    amI ( ) ;
    • • •
  }
}

```

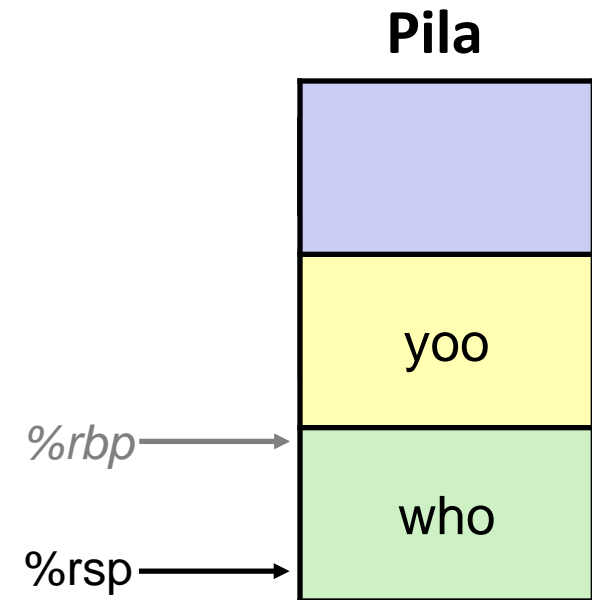
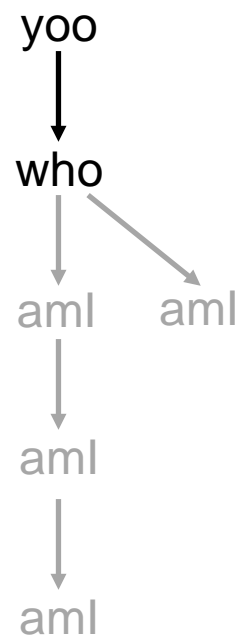
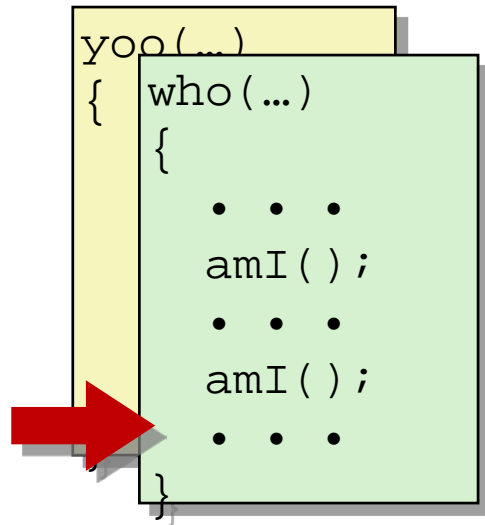
A red arrow points from the left towards the nested function calls. The diagram illustrates a sequence of function calls: `yoo(...)` calls `who(...)`, which in turn calls `amI()` twice. The function calls are represented by nested boxes: a yellow box for `yoo(...)` and a green box for `who(...)`. The `amI()` calls are represented by three dots and the function name.



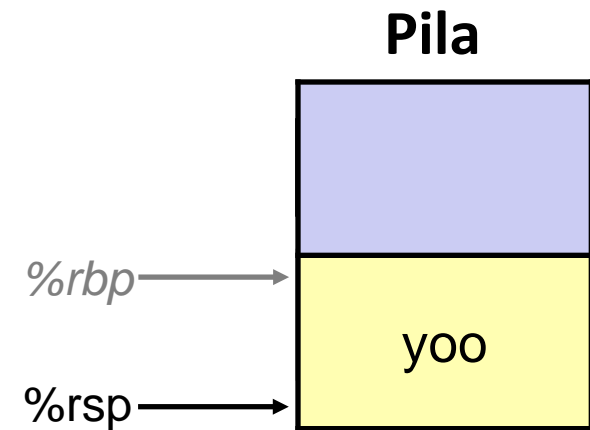
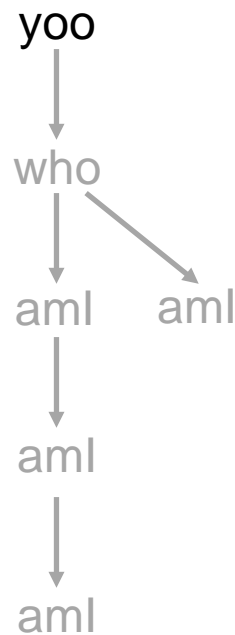
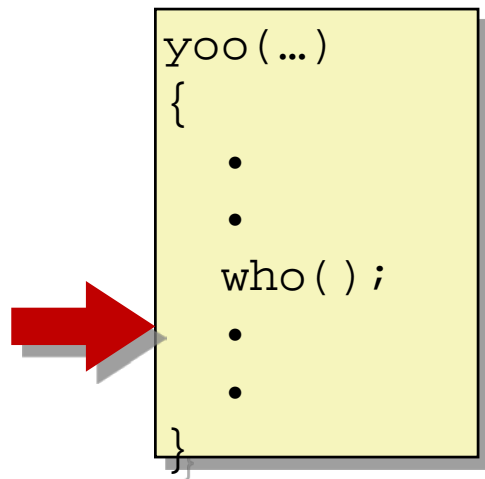
Ejemplo



Ejemplo



Ejemplo



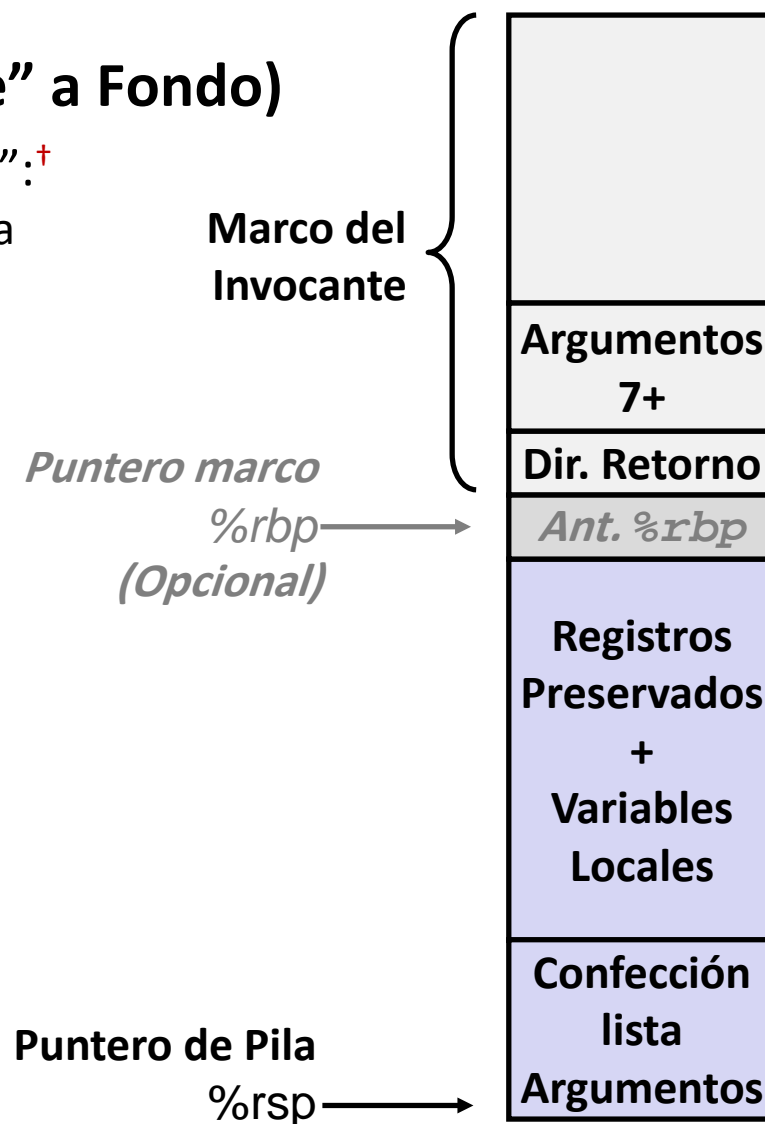
Marco de Pila x86-64/Linux

■ Contenidos Marco Pila (de “Tope” a Fondo)

- “Confección de la lista de argumentos”:[†]
Parámetros (7+) función punto ser llamada
- Variables locales
Si no se pueden mantener en registros
- Contexto registros preservados
- *Antiguo puntero de marco (opcional)*
usando -fno-omit-frame-pointer (ó -O0)
 - *Dir.retorno pertenece marco anterior*

■ Marco de Pila del Invocante

- Dirección de retorno
 - Salvada por la instrucción call
- Argumentos (7+) para esta llamada



Ejemplo: incr

```
long incr(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

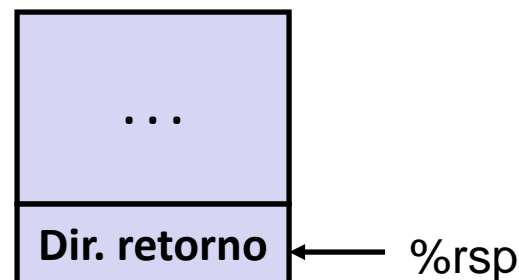
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Registro	Uso(s)
%rdi	Argumento p
%rsi	Argumento val , y
%rax	x , Valor de retorno

Ejemplo: llamar a incr #1

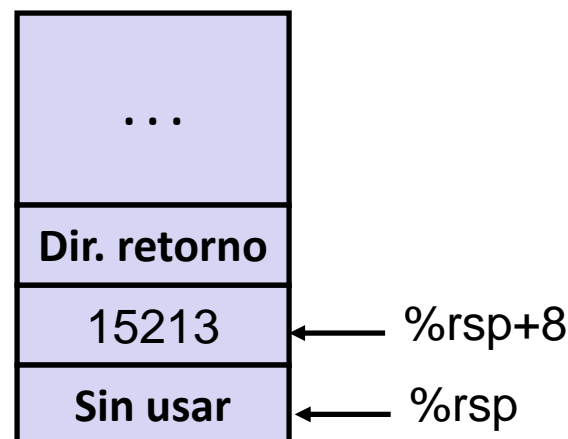
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Estructura de Pila inicial



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Estructura de Pila resultante

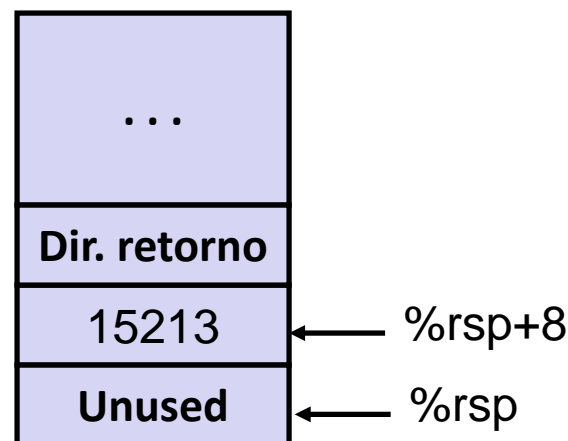


Ejemplo: llamar a incr #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Estructura de Pila



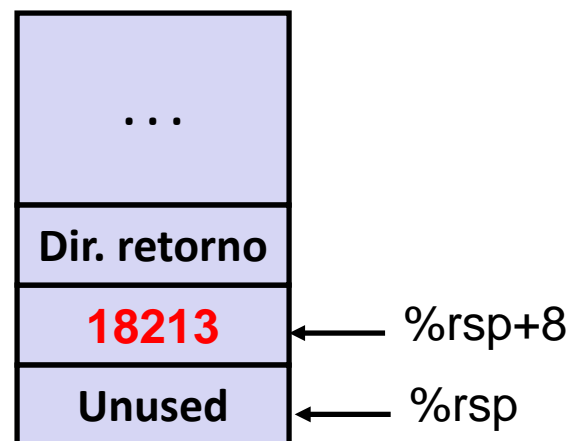
Registro	Uso(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	<code>3000</code>

Ejemplo: llamar a incr #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Estructura de Pila

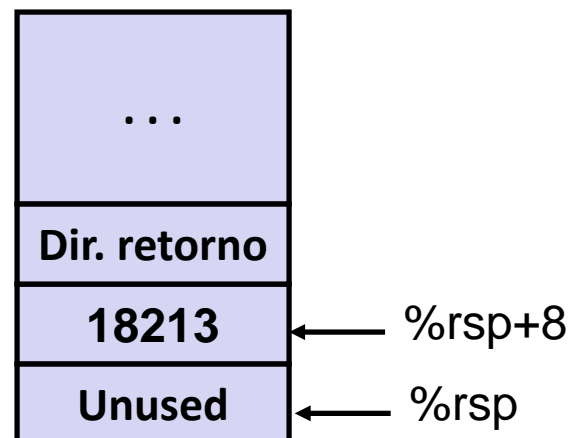


Registro	Uso(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	<code>3000</code>

Ejemplo: llamar a incr #4

Estructura de Pila

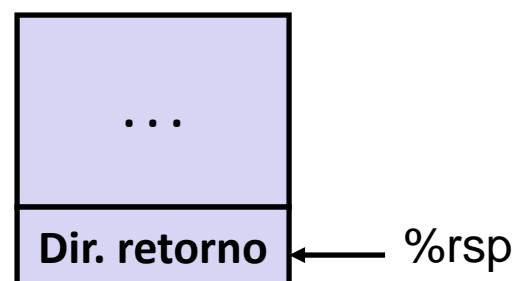
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Registro	Uso(s)
%rax	Valor de retorno

Estructura de Pila resultante

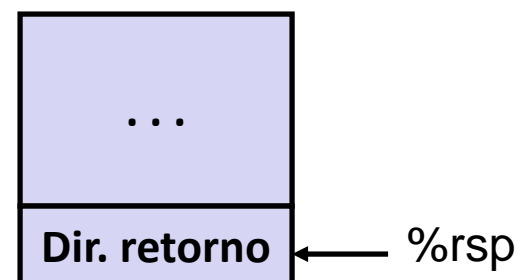


Ejemplo: llamar a incr #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

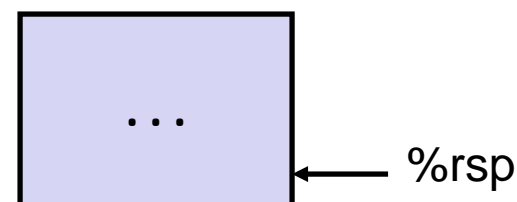
```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Estructura de Pila resultante



Registro	Uso(s)
%rax	Valor de retorno

Estructura de Pila final



Convenciones de Preservación de Registros[†]

- Cuando el procedimiento yoo llama a who:
 - yoo es el *que llama (invocante, llamante)*
 - who es el *llamado (invocado)*
- ¿Se puede usar un registro para almacenamiento temporal?

```
yoo:
    . . .
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    . . .
    ret
```

```
who:
    . . .
    subq $18213, %rdx
    . . .
    ret
```

- Contenidos del registro %rdx sobrescritos por who
- Podría causar problemas → ¡debería hacerse algo!
 - Necesita alguna coordinación

Convenciones de Preservación de Registros

- Cuando el procedimiento *yoo* llama a *who*:
 - *yoo* es el *que llama (invocante, llamante)*
 - *who* es el *llamado (invocado)*
- ¿Se puede usar un registro para almacenamiento temporal?
- Convenciones[†]
 - “*Salva Invocante*”
 - El que llama salva valores temporales en su marco antes de la llamada
 - “*Salva Invocado*”
 - El llamado salva valores temporales en su marco antes de usar (regs.)
 - ...y los restaura antes de retornar al que llama

Uso de Registros en Linux x86-64[†] #1

■ %rax

- Valor de retorno
- También salva-invocante
- *Puede ser modificado[‡] por el proc.*

■ %rdi, ..., %r9

- Argumentos[‡]
- También salva-invocante
- Pueden ser modificados[‡] por proc.

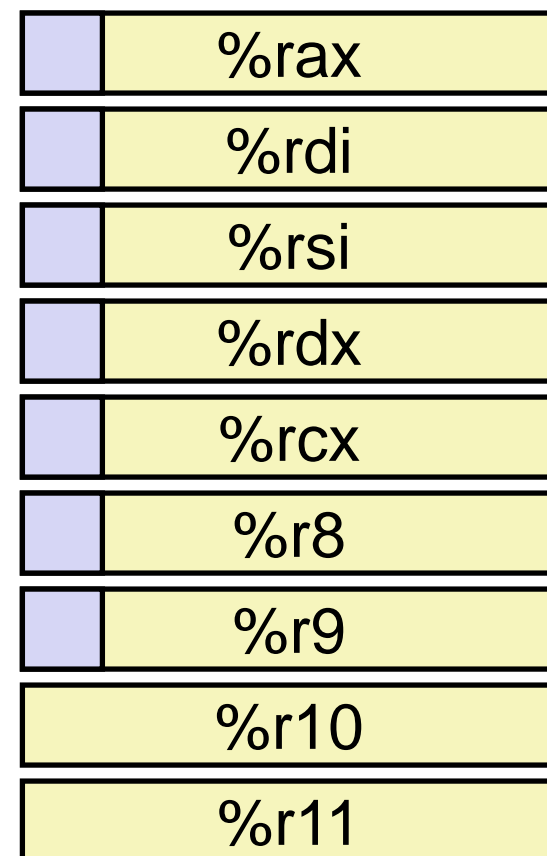
■ %r10, %r11

- Salva-invocante
- Pueden ser modificados[‡] por proc.

Val.retorno
S-Invocante

Argumentos
S-Invocante

Temporales
S-Invocante



[†] Ver Wikipedia: "X86 calling conventions", Sys5 AMD64 ABI
[‡] regla mnemotécnica: invocado-cuidado, invocante-adelante
[‡] regla mnemotécnica: Diane's Silk Dress Costs \$89

Uso de Registros en Linux x86-64 #2

■ **%rbx, %r12 - %r15**

- Salva-invocado
- Invocado debe preservar y restaurar

■ **%rbp**

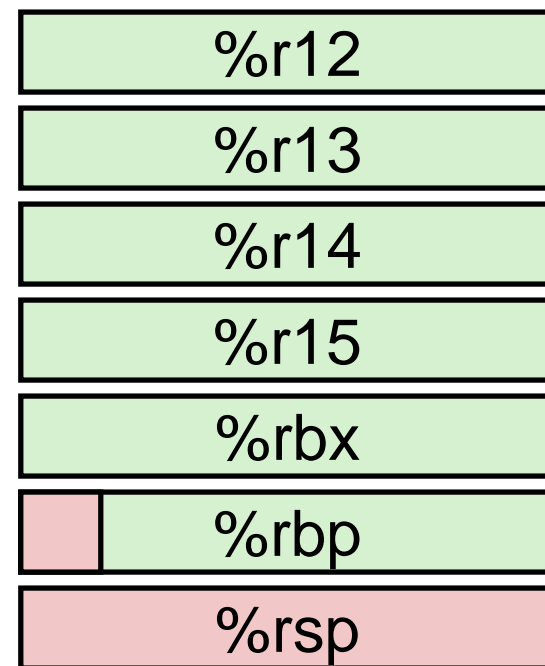
- Salva-invocado
- Invocado debe preservar y restaurar
- Puede que se use como marco pila
- Puede usarse intermezcladamente[†]

■ **%rsp**

- Forma especial de salva-invocado
- Restaurado a su valor original a la salida del procedimiento

Temporales
S-Invocado

Especiales



[†] "mix & match", se refiere a que podrían linkarse procedimientos compilados con gcc -fno-omit... y -fomit... y no pasaría nada porque %rbp seguiría siendo s-invocado

Mini-Ejercicio

```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4,
           long a5, long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4)+
           add5(a5, a6, a7, a8, a9);
}
```

■ ¿Dónde se pasan a0,..., a9?

rdi, rsi, rdx, rcx, r8, r9, pila

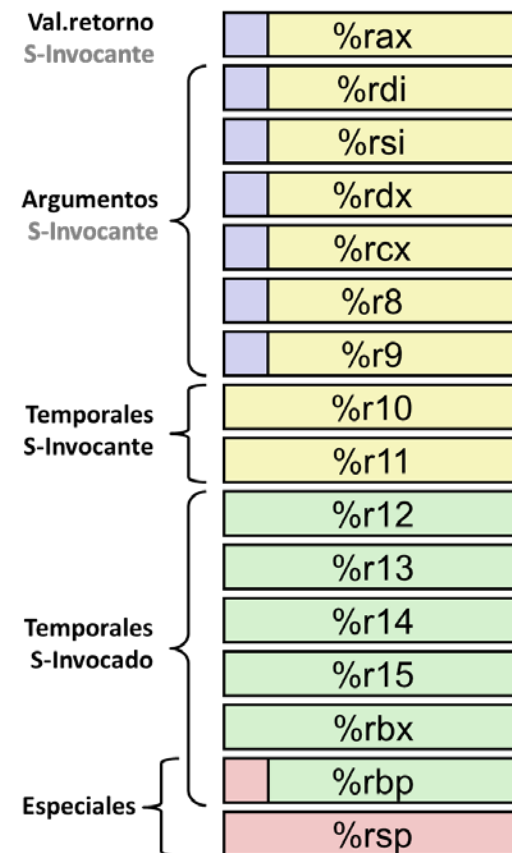
■ ¿Dónde se pasan b0,..., b4?

rdi, rsi, rdx, rcx, r8

■ ¿Qué registros tenemos que preservar?

Pregunta mal formulada. Requiere ver ASM.

rbx, rbp, r9 (durante 1ª llamada a add5)



Mini-Ejercicio

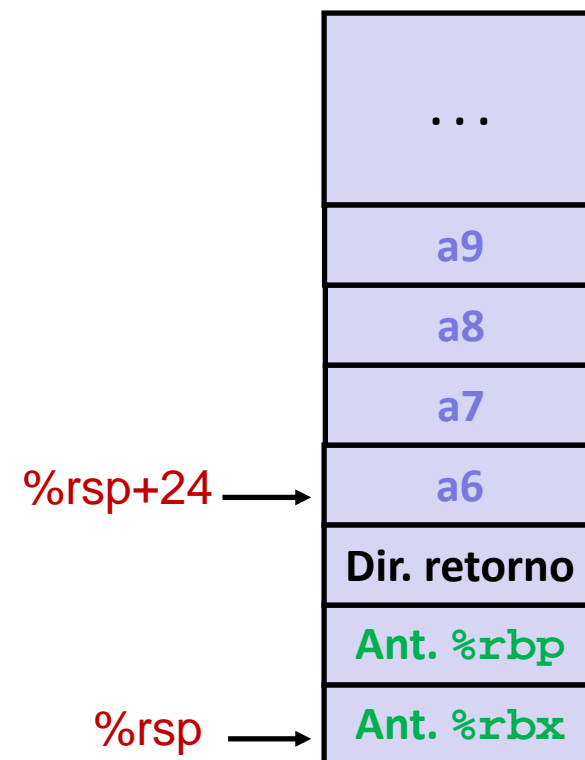
```
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4,
           long a5, long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4)+
           add5(a5, a6, a7, a8, a9);
}
```

```
add10:
    pushq    %rbp
    pushq    %rbx
    movq     %r9, %rbp
    call     add5
    movq     %rax, %rbx
    movq     48(%rsp), %r8
    movq     40(%rsp), %rcx
    movq     32(%rsp), %rdx
    movq     24(%rsp), %rsi
    movq     %rbp, %rdi
    call     add5
    addq     %rbx, %rax
    popq     %rbx
    popq     %rbp
    ret
```

```
add5:
    addq     %rsi, %rdi
    addq     %rdi, %rdx
    addq     %rdx, %rcx
    leaq     (%rcx,%r8), %rax
    ret
```

Pila durante
la llamada a
add10

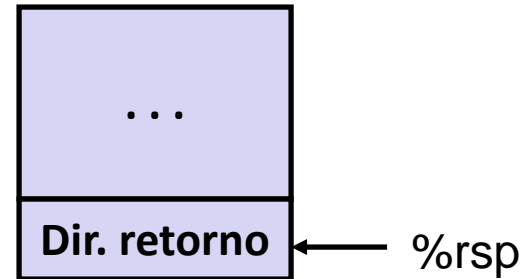


Ejemplo Salva-invocado #1

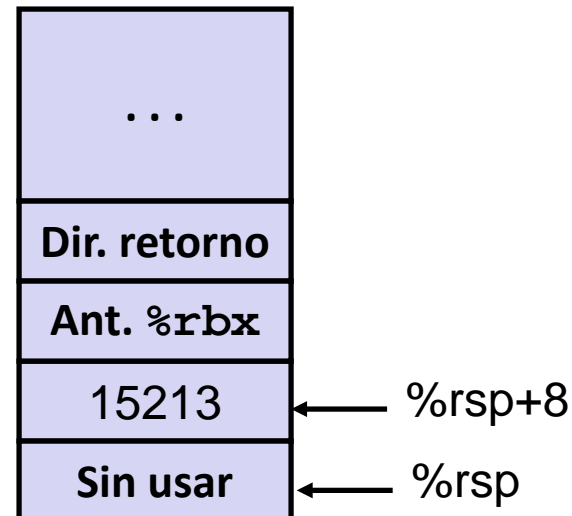
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq    %rbx
    subq     $16, %rsp
    movq     %rdi, %rbx
    movq     $15213, 8(%rsp)
    movl     $3000, %esi
    leaq     8(%rsp), %rdi
    call     incr
    addq     %rbx, %rax
    addq     $16, %rsp
    popq     %rbx
    ret
```

Estructura de Pila inicial



Estructura de Pila resultante

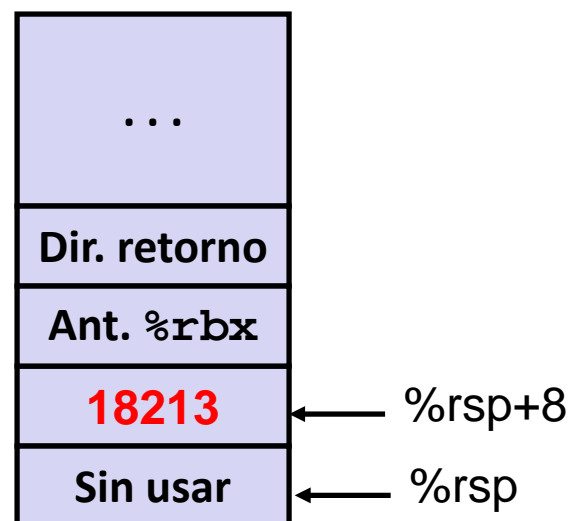


Ejemplo Salva-invocado #2

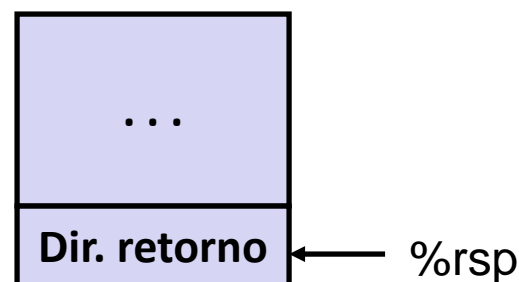
Estructura de Pila

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
    pushq    %rbx
    subq     $16, %rsp
    movq     %rdi, %rbx
    movq     $15213, 8(%rsp)
    movl     $3000, %esi
    leaq     8(%rsp), %rdi
    call     incr
    addq     %rbx, %rax
    addq     $16, %rsp
    popq     %rbx
    ret
```



Estructura de Pila antes de ret



Programación Máquina III: Procedimientos

- **Procedimientos**
 - Mecanismos
 - Estructura de la pila
 - Convenciones de llamada
 - Pasando el control
 - Pasando los datos
 - Gestionando datos locales
 - **Ejemplos ilustrativos de Recursividad**

Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Condición Terminación Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

```
.L6:
    rep; ret
```

Registro	Usos	Tipo
%rdi	x	Salva-invocante <input type="checkbox"/>
%rax	Valor de retorno	Salva-invocante <input type="checkbox"/>

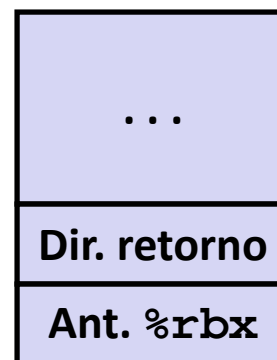
Preservar Registro para Llamada Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

rep; ret



Registro	Usos(s)	Tipo
%rbx	x & 1	Salva-invocado <input type="checkbox"/>

Preparar Llamada Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```



```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Registro	Usos	Tipo
%rbx	x & 1	Salva-invocado <input type="checkbox"/>
%rdi	x >> 1 Argumento recurs.	Salva-invocante <input type="checkbox"/>

Llamada Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Registro	Usos	Tipo
%rbx	x & 1	Salva-invocado 
%rax	Valor de retorno de llamada recursiva	Salva-invocante 

Resultado Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Registro	Usos(s)	Tipo
%rbx	x & 1	Salva-invocado <input type="checkbox"/>
%rax	Valor de retorno	Salva-invocante <input type="checkbox"/>

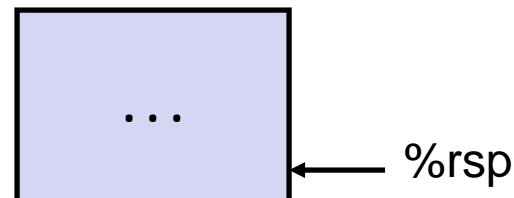
Terminar Función Recursiva

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

rep; ret



Registro	Usos	Tipo
%rax	Valor de retorno	Salva-invocante <input type="checkbox"/>

Observaciones Sobre la Recursividad

■ Manejada sin Especiales Consideraciones

- Marcos pila implican que cada llamada a función tiene almcnmto. privado
 - Variables locales y Registros preservados
 - Dirección de retorno salvada
- Convenciones preservación registros previenen que una llamada a función corrompa los datos de otra
 - A menos que el código C explícitamente lo haga (p.ej. buffer overflow)
- Disciplina de pila sigue el patrón de llamadas / retornos
 - Si P llama a Q, entonces Q retorna antes que P
 - Primero en entrar, último en salir (Last-In, First-Out)[†]

■ También funciona con recursividad mutua[‡]

- P llama a Q; Q llama a P

[‡] en general con reentrancia

[†] pila = lista LIFO

Resumen de Procedimientos en x86-64

■ Puntos Importantes

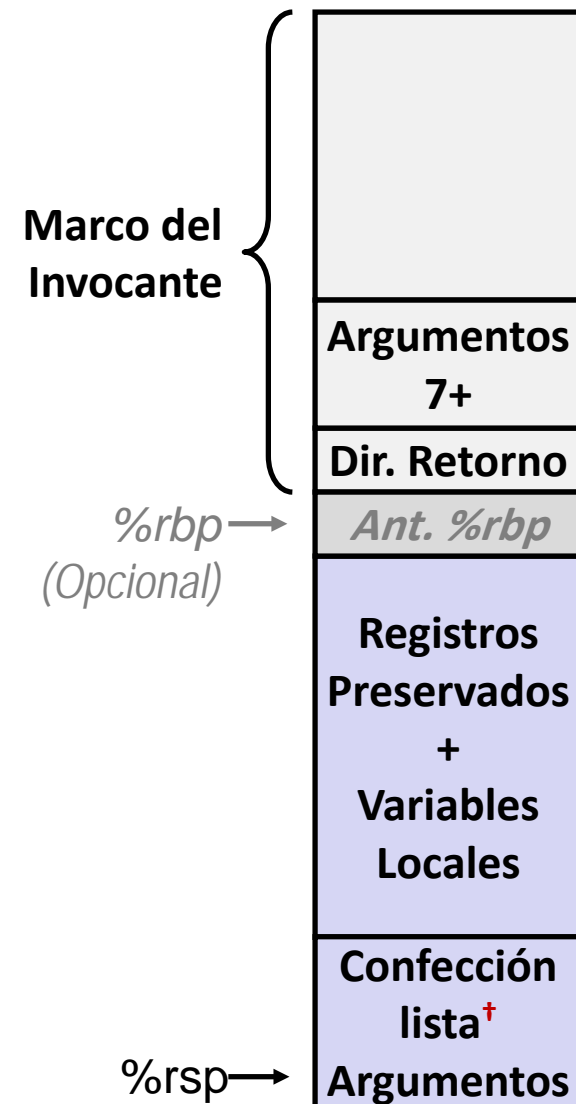
- Pila es la estructura de datos correcta para llamada / retorno procedimientos
 - P llama a Q, entonces Q retorna antes que P

■ Recursividad (y recursividad mutua) con mismas convenciones de llamada normales

- Se pueden almacenar valores tranquilamente en el marco de pila local y en registros salva-invocado
- Poner argumentos 7+ de la función en tope de pila
- Devolver resultado en %rax

■ Punteros son direcciones de valores

- Global o en pila



Guía de trabajo autónomo (4h/s)

■ **Estudio:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Procedures
 - § 3.7 pp.274-291
- Understanding Pointers, Using GDB.
 - § 3.10.1-2 pp.312-316

■ **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.32 § 3.7.2, pp.280
- Probl. 3.33 § 3.7.3, pp.282
- Probl. 3.34 § 3.7.5, pp.288
- Probl. 3.35 § 3.7.6, pp.290

Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIIT/[C.1 BRY com](#)