EC 21-22: programación a nivel máquina

Tema 2.1: conceptos básicos y aritmética

1. Historia de los procesadores y arquitecturas

Los procesadores Intel x86 dominan el mercado de portátiles, sobremesas y servidores. Diseño evolutivo, se van añadiendo funcionalidades sobre el 8086 (introducido en 1978).

Es un computador CISC:

- Muchas instrucciones distintas, con muchos formatos distintos
  - Sólo un pequeño subconjunto aparece en programas Linux
- Era difícil igualar las prestaciones de un repertorio RISC, pero Intel lo ha conseguido (en lo que a velocidad se refiere, no en bajo consumo)

Evolución x86: hitos

Nombre	Año	Transistores	MHz	Descripción
8086	1978	29K	5-10	Primer procesador Intel 16bit. Espacio de direccionamiento 1MB.
386	1985	275K	16-33	Primer procesador Intel 32bit de la familia IA32 (x86). Direccionamiento plano.
Pentium 4E	2004	125M	2800-3800	Primer procesador Intel 64bit de la familia x86
Core 2	2006	291M	1060-3500	Primero procesador Intel multicore
Core i7	2008	731M	1700-3900	Cuatro cores, hyperthreading (2 vías)

Clones x86: AMD

Históricamente, AMD ha ido siguiendo a Intel en todo, con CPUs ligeramente más lentas pero mucho más asequibles. En cierto punto contrataron a grandes diseñadores de circuitos y desarrollaron Opteron, un duro competidor para el Pentium 4. También desarrollaron x86-64, su propia extensión de 64bits.

Sin embargo, a día de hoy Intel sigue liderando el sector debido a su inversión en I+D en tecnología de semiconductores, la cual AMD subcontrata.

Historia del 64bit de Intel

- 2001: Intel intenta un cambio radical de IA32 a IA64
  - Arquitectura distinta
  - Ejecuta codigo IA32 sólo como legacy
  - Prestaciones decepcionantes
- 2003: AMD interviene con una solución evolutiva, x86-64 (ahora llamado AMD64)

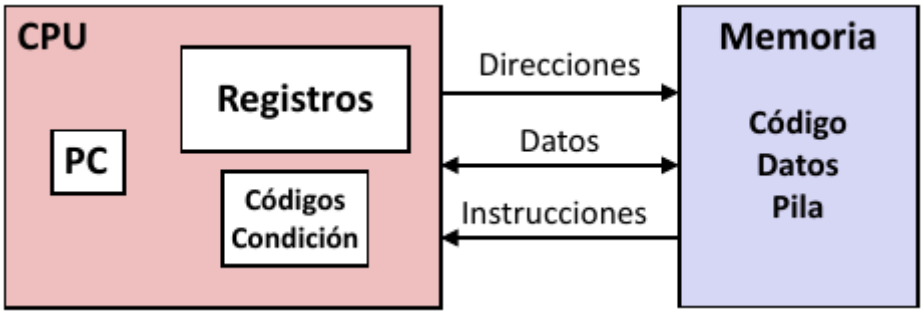
    - Intel se ve obligada a concentrarse y mejorar IA64 para no admitir que AMD ha hecho algo mejor
- 2004: Intel anuncia extensión EM64T de IA32 (ahora llamada Intel64)
    - Extended Memory 64bit Tech
    - Casi igual a x86-64 de AMD
- Todos los procesadores x86 salvo gamas bajas soportan el x86-64, gran parte del código sigue siendo 32bit

2. Lenguaje C, ensamblador y código máquina

Definiciones

- **Arquitectura (también ISA):** las partes del diseño de un procesador que se necesitan entender para escribir código ensamblador (especificación del registro de instrucciones, registros...)
- **Formas de código:**
    - Código máquina: programas (codops, bytes) que ejecuta el procesador
    - Código ensamblador: representación textual del código máquina
- **Microarquitectura:** implementación de la arquitectura (tamaño de la caché, frecuencia del core...)

Perspectiva código ensamblador / código máquina

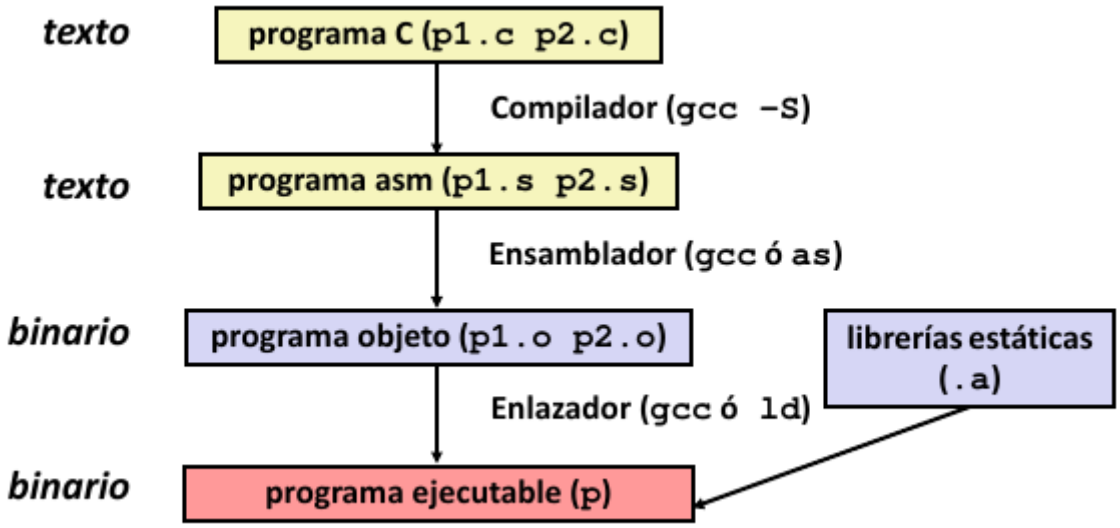


Estado visible al programador:

- **PC - Contador del programa:** dirección de la próxima instrucción, RIP en x86-64
- **Archivo de registros:** datos del programa muy utilizados
- **Códigos de condición / flags de estado:** almacenan información sobre la operación aritmético-lógica más reciente. Usados para bifurcación condicional
- **Memoria:** array direccionable por bytes, contiene código y datos de usuario. Soporte de pila a procedimientos

Convertir C en código objeto

> - Código en ficheros `p1.c p2.c`
> - Compilación con `gcc -0g p1.c p2.c -o p`



Representación de datos C, IA32, x86-64

Tipo de dato C	Normal 32b	IA32	x86-64
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char * (cualquier puntero)	4	4	8

Características ensamblador

Tipos de datos

- **Datos enteros** de 1,2,4,8 bytes
  - Valores de datos
  - Direcciones (punteros sin tipo)
- **Datos en coma flotante** de 4,8,10 bytes
- **Código**: secuencias de bytes codificando serie de instrucciones
- **No hay tipos compuestos** como arrays o estructuras, sólo bytes posicionados contiguamente en memoria

Instrucciones

- **Operaciones:** función aritmético-lógica sobre datos en registros o memoria
- **Instrucciones de transferencia:** transferencia de datos entre memoria y registros
  - Cargar datos de memoria a registro
  - Almacenar datos de registro a memoria

- **Instrucciones de control:** transferencia de contol
    - Incondicionales: saltos, llamadas a procedimientos, retorno desde un proceso
    - Saltos condicionales

Código objeto

Ensamblador

- Pasa de .s (ensamblador) a .o (objeto)
- Instrucciones codificadas en binario
- Imagen casi completa del código ejecutable
- Le faltan enlaces entre código de ficheros diferentes

Enlazador

- Resuelve diferencias entre ficheros
- Combina con librerías de tiempo ejecuciones estáticas (ej. código con malloc o printf)
- Algunas librerías son enlazadas dinámicamente
    - El enlace ocurre cuando el programa comienza la ejecución

Ejemplo de instrucción máquina

Código C: almacenar el valor t donde indica (apunta) dest

```
*dest = t;
```

Ensamblador: mover un valor de 8B (palabra quad) a memoria.

- t: registro %rax
- dest: registro %rbx
- *dest: memoria M[%rbx]

```
movq %rax, (%rbx)
```

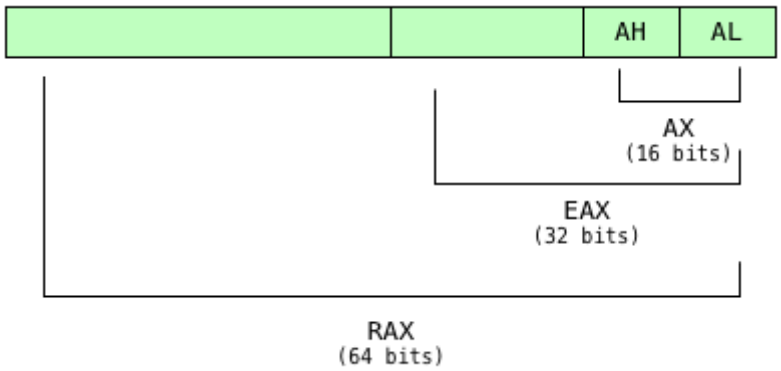
Código objeto: instrucción de 3Bytes almacenada en dirección 0x40059e

```
0x40059e: 48 89 03
```

3. Conceptos básicos asm: registros, operandos y move

Registros enteros x86-64

16 registros enteros: Diane's Silk Dress Cost 89 + a, b + rsp,rbp + 10 al 15



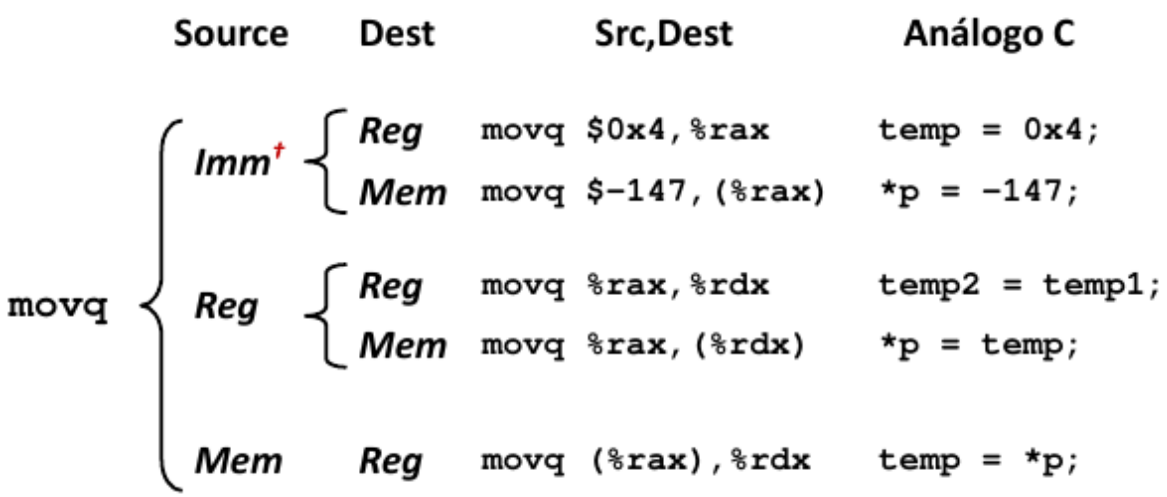
- Diane's Silk Dress Cost 89 para los argumentos de funciones
- %rax para return de funciones
- %rsp es puntero de pila, no se usa para guardar datos
- %ebp puntero base
- Registros de propósito general: %eax, %ebx, %ecx, %edx, %esi, %edi
- Compatibilidad ascendente (lo que guardo en %eax lo puedo guardar en %rax, aunque esté malgastando. Al revés no)

Mover datos

```
movq src, dst
```

Tipos de operandos:

- Inmediatos: datos enteros constantes, codificados mediante 1, 2 o 4 bytes
  - `movq $0x400, $123456`
- Registro: alguno de los 16 registros enteros, menos %rsp y otros
  - `movq %rax, %r13`
- Memoria: 8 bytes consecutivos memoria en dirección dada por un registro
  - No se puede transferir memoria a memoria en una sola instrucción
  - `movq (%rax), %rdx`



Modos direccionamiento a memoria sencillos

- **Normal/Indirecto a través de registro:**
  - `(R)` `Mem[Reg[R]]`
  - El registro R indica la dirección de memoria
  - Igual que seguir un puntero en C
    - `movq (%rcx), %rax`
- **Desplazamiento:**
  - `D(R)` `Mem[Reg[R]+D]`
  - El registro R indica el inicio de una región de memoria
  - La D el desplazamiento
    - `movq 8(%rbp), %rdx`

Modos direccionamiento a memoria completos

- **Forma general:**
  - `D(Rb, Ri, S)` `Mem[Reg[Rb] + S*Reg[Ri] + D]`
  - D: desplazamiento, constante 1, 2 o 4 Bytes
  - Rb: registro base, cualquiera de los 16 registros enteros
  - Ri: registro índice, cualquiera menos %rsp
  - S: factor de escala, 1, 2, 4 o 8
- **Casos especiales:**
  - `(Rb, Ri)` `Mem[Reg[Rb] + Reg[Ri]]`
  - `D(Rb, Ri)` `Mem[Reg[Rb] + Reg[Ri] + D]`
  - `(Rb, Ri, S)` `Mem[Reg[Rb] + S*Reg[Ri]]`

Ejemplos cálculo de direcciones

Dado lo siguiente:

%rdx	**0xF000**
%rcx	**0x0100**

Expresión	Cálculo de dirección	Dirección
0x8(%rdx)	0xF000 + 0x0008	0xF008
(%rdx, %rcx)	0xF000 + 0x0100	0xF100
(%rdx,%rcx,4)	0xF000 + 0x0100 * 4	0xF400
0x80(,%rdx,2)	0 + 0x0080 + 0xF000 * 2	0x1E080

4. Operaciones aritmético-lógicas

Instrucción para el cálculo de direcciones

```
leaq src, dest
```

Donde src es cualquier expresión de modo de direccionamiento a memoria. Ajusta Dest a la dirección indicada por la expresión.

Usos:

- Calcular direcciones sin hacer referencias a memoria
- Calcular expresiones aritméticas de la forma x+k*y, donde k=1,2,4 u 8

```
long m12(long x)
{
  return x*12;
}
```

Traducción a ASM por el compilador:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Algunas operaciones con dos operandos

- Aritméticas:
  - addq src,dest dest=dest+src
  - subq src,dest dest=dest-src
  - imulq src,dest dest=dest*src
  - salq src,dest dest=dest<<src
  - sarq src,dest dest=dest>>src
- Lógicas:
  - shrq src,dest dest=dest>>src
  - xorq src,dest dest=dest^src
  - andq src,dest dest=dest&src
  - orq src,dest dest=dest|src

Algunas operaciones con un operando

- Aritméticas:
  - incq dest dest++
  - decq dest dest--
  - negq dest -dest
- Lógicas
  - notq dest dest=not dest

Tema 2.2: control

1. Control: códigos de condición

Estado del procesador

Información sobre el programa ejecutándose actualmente. Incluye:

- Datos temporales (registros RPG)
- Situación de la pila en tiempo de ejecución (%rsp)
- Situación actual del contador de programa (%rip)
- Estado de comparaciones recientes (flags, códigos de condición)

Códigos de condición

Registros de un solo bit:

- CF, carry flag, flag de acarreo
- ZF, zero flag, flag de cero
- SF, sign flag, flag de signo
- OF, overflow flag

Se ajustan implícitamente por las operaciones aritméticas. No son afectados por la instrucción lea. Pongamos por ejemplo `addq src,dest` (t=a+b):

- CF a 1, accarreo del bit más significativo, desbordamiento operación sin signo
- ZF a 1, t==0
- SF a 1, t<0 en operaciones con signo
- OF a 1, desbordamiento operación con signo

También podemos ajustarlos explícitamente con la instrucción compare (`cmpq a, b`), la cual realmente equivale a b-a pero sin guardar el resultado en b. En tal caso:

- CF a 1, acarreo del bit más significativo, operaciones sin signo
- ZF a 1, a=b
- SF a 1, b-a < 0 en operaciones con signo
- OF a 1, desbordamiento en operación con signo
    - OF = Cn^Cn-1
    - CF = Cn

Otra opción de ajuste explícito lógico es la instrucción `testq a,b`, que en realidad hace a&b pero sin guardar el resultado en b. En tal caso:

- ZF a 1, a&b == 0 (alguno de los 2 es 0)
- SF a 1, a&b < 0

Para saber si un valor es 0: `cmpq $0, %rax;  testq %rax, %rax`

Consultar códigos de condición

Las instrucciones `SetCC dest` sirven para ajustar el byte de destino a 0/1 según el código de condición indicado con CC (la combinación de flags que queramos).

Dst registro debe ser tamaño Byte, Dst memoria solo se modifica el B menos significativo

SetCC	Condición	Descripción
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Sign (negativo)
setns	~SF	Not Sign
setg	~(SF^OF)&~ZF	Greater (signo)
setge	~(SF^OF)	Greater or Equal (signo)
setl	(SF^OF)	Less (signo)
setle	(SF^OF)|ZF	Less or Equal (signo)
seta	~CF&~ZF	Above (sin signo)
setb	CF	Below (sin signo)

Las operaciones que modifican un registro de 32 bits, ponen a 0 el resto hasta 64 bits

2. Saltos condicionales

Las instrucciones `jCC` saltan a otro punto del código si se cumple el código de condición CC.

jCC	Condición	Descripción
jmp	1	Incondicional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Sign (negativo)
jns	~SF	Not Sign
jg	~(SF^OF)&~ZF	Greater (signo)
jge	~(SF^OF)	Greater or Equal (signo)
jl	(SF^OF)	Less (signo)
jle	(SF^OF)|ZF	Less or Equal (signo)
ja	~CF&~ZF	Above (sin signo)
jb	CF	Below (sin signo)

Ejemplo salto condicional a la antigua

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
absdiff:
   cmpq    %rsi, %rdi   # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:                    # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

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

En este caso, la parte del else es una función a parte a la que saltamos si se cumple la condición (jle, jump less or equal)

Expresándolo con código goto

C permite la sentencia goto, en la que salta a la posición indicada por la etiqueta:

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Traducción en General Expresión Condicional (con saltos)

Código C:

```
val = test? then_expr : else_expr

ej:
    val = x>y? x-y; y-x;
```

Versión goto:

```
nottest = !test
if  (nottest) goto Else
val = then_expr;
goto Done;

Else:
    val=else_expr;

Done:
    ---
```

Creamos regiones de código separadas para las expresiones then y else, ejecutamos la adecuada.

Usando movimientos condicionales

Las instrucciones implementan un `if (test) dest<-src` en procesadores posteriores a 1995 (Pentium Pro/II). GCC intenta utilizarlas, pero sólo cuando sepa que es seguro.

Esto se debe a que puede tener ramificaciones muy perjudiciales al flujo de instrucciones en cauces. Pero es bueno usarlos porque el movimiento condicional no requiere transferencia de control. Casos en los que es mejor no usarlo:

- **Cálculos costosos:** sólo tiene sentido no hacer transferencia a UC cuando son cálculos muy sencillos

- `val = Test(x) ? Hard1(x) : Hard2(x);`

- **Cálculos arriesgados:** puede tener efectos no deseables
  - `val = p ? *p : 0;`
- **Cálculos con efectos colaterales:** no debería haberlos
  - `val = x > 0 ? x*=7 : x+=3;`

Recordamos que se calculan ambos valores

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Registro	Uso(s)
%rdi	Argumento **x**
%rsi	Argumento **y**
%rax	Valor de retorno

```
absdiff:
    movq    %rdi, %rax  # x
    subq    %rsi, %rax  # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx  # eval = y-x
    cmpq    %rsi, %rdi  # x:y
    cmovle  %rdx, %rax  # if <=, result = eval
    ret
```

† generar con gcc -fif-conversion -Og -S c
ó incluso con acc -O -S c

3. Bucles

Traducción general del do-while:

Código C:

```
do
    /*body*/
while (test)
```

Versión goto:

```
loop:
    /*body*/
    if(test)
        goto(loop)
```

Ejemplo do-while

El siguiente código es el popcount (cuenta el número de unos del argumento dado). Usa salto condicional para seguir iterando o salir del bucle.

Código C

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Versión Goto

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
†  rep; ret
```

† problema predicción saltos Opteron y Ath
Software Optimization Guide for AMD64 Fa

Traducción general del while #1, -Og -O0:

Código C:

```
while (test)
    /*body*/
```

Versión goto:

```
goto test;

loop:
    /*body*/

test:
    if(test)
        goto loop

done:
    ---
```

Traducción del tipo "jump-to-middle", usada con -O0/-Og

Ejemplo while #1

Código C

```
long pcount_while
   (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Salta-en-medio†

```
long pcount_goto_jtm
   (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

En este caso el goto inicial salta a test (jump-to-middle).

Traducción general del while #2, -O1:

Versión while:

```
while (test)
    /*body*/
```

Versión do-while:

```
if(!test)
    goto done;
do
    /*body*/
    while(test)

done:
    ---
```

Versión goto:

```
if(!test)
    goto done;

loop:
    /*body*/
    if(test)
        goto loop;

done:
    ---
```

Traducción tipo "copia-test", conversión a do-while. Usada con -O1.

Código C

```
long pcount_while
   (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Copia-test

```
long pcount_goto_ct
   (unsigned long x) {
  long result = 0;
  if (!x) goto done;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
 done:
  return result;
}
```

Forma del bucle for, conversión a while

```
for (Init; Test; Update)
    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Versión While

```
Init;
while (Test) {
    Body
    Update;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

Conversión a do-while

Código C

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

- **La comprobación inicial se puede optimizar (quitándola)**

Versión Goto

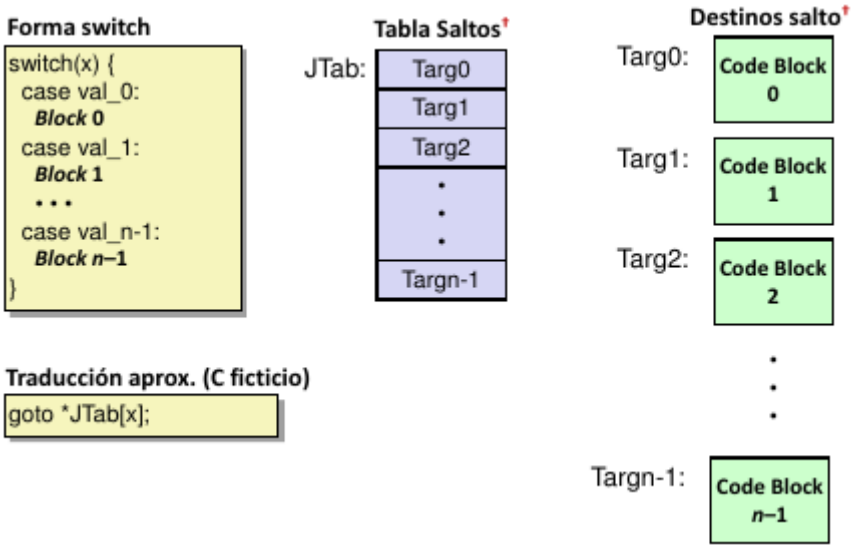
```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;             Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

4. Sentencias switch

- Múltiples etiquetas de caso (casos 1,2,3,5,6)
- Caídas en cascada / fallthrough cuando no ponemos break (el código continua con el siguiente caso hasta encontrar un break, puede usarse a nuestro favor) (caso 2)
- Casos ausentes (caso 4)

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
       /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Estructura de una tabla de saltos



Ejemplo sentencia switch

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

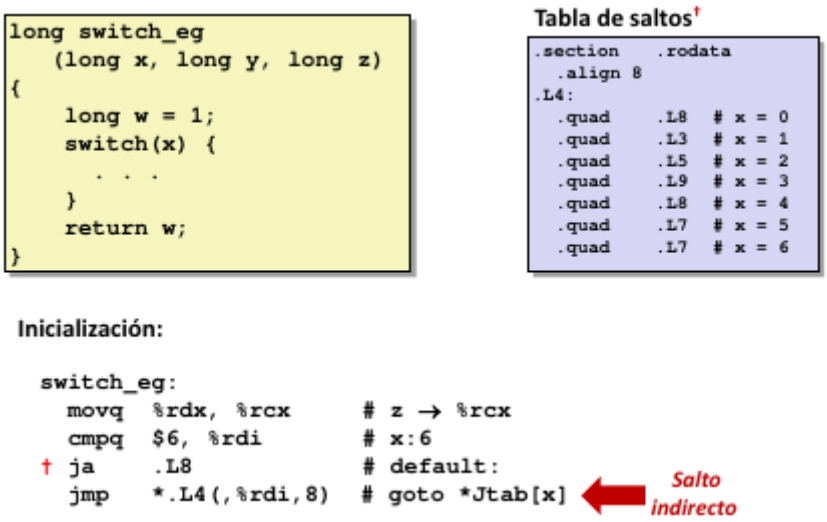
Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rdx	Argumento z
%rax	Valor de retorno

Inicialización:

```
switch_eg:
   movq   %rdx, %rcx      # z → %rcx
   cmpq   $6, %rdi        # x:6
† ja     .L8              # default:
   jmp    *.L4(,%rdi,8)   # goto *Jtab[x]
```

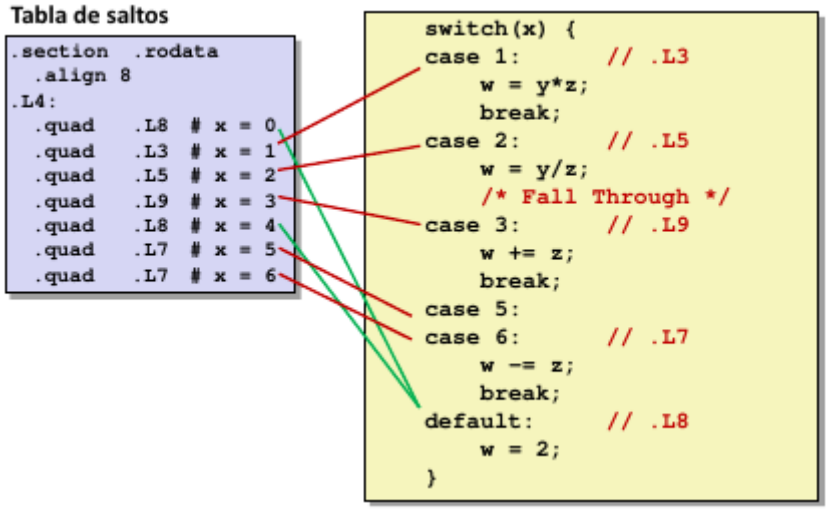
Notar que **w** no se inicializa aquí

¿Qué rango de valores cubre default?



Explicación inicialización ensamblador

- **Estructura de la tabla:** cada destino salto requiere 8B. La dirección base es .L4
- **Saltos:**
  - **Directo:** `jmp .L8`, destino de salto indicado por la etiqueta .L8
  - **Indirecto:** `jmp *.L4(,%rdi,8)`
    - Inicio de la tabla de saltos: .L4
    - Se escala por un factor de 8 (las direcciones ocupan 8B)
    - Captar destino salto desde la dirección efectiva: .L4 + (x*8), para 0<=x<=6



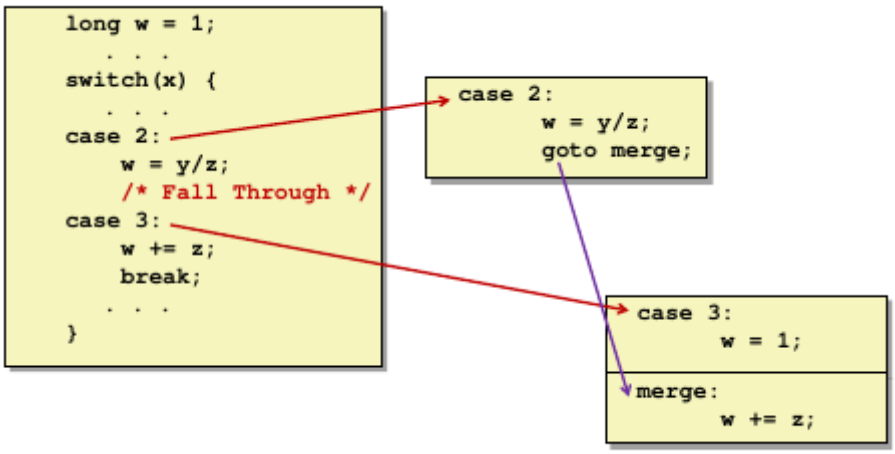
Bloques de código: x==1

```
switch(x) {
case 1:     // .L3
    w = y*z;
    break;
. . .
}
```

```
.L3:
   movq    %rsi, %rax  # y
   imulq   %rdx, %rax  # y*z
   ret
```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rdx	Argumento z
%rax	Valor de retorno

Bloques de código: x==2, x==3, caída en cascada




```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
   . . .
}
```

```
.L5:                    # Case 2
   movq    %rsi, %rax
†  cqto
   idivq   %rcx        #  y/z
   jmp     .L6         #  goto merge
.L9:                    # Case 3
   movl    $1, %eax    #  w = 1
.L6:                    # merge:
   addq    %rcx, %rax  #  w += z
   ret
```

Registro	Uso(s)
%rdi	Argumento x
%rsi	Argumento y
%rdx	Argumento z
%rax	Valor de retorno

Bloques de código: x==5, x==6

```
switch(x) {
. . .
case 5:  // .L7
case 6:  // .L7
    w -= z;
    break;
default: // .L8
    w = 2;
}
```

```
.L7:                # Case 5,6
   movl  $1, %eax   #  w = 1
   subq  %rdx, %rax #  w -= z
   ret
.L8:                # Default:
   movl  $2, %eax   #  2
   ret
```

Registro	Uso(s)
`%rdi`	Argumento `x`
`%rsi`	Argumento `y`
`%rdx`	Argumento `z`
`%rax`	Valor de retorno

Resumen

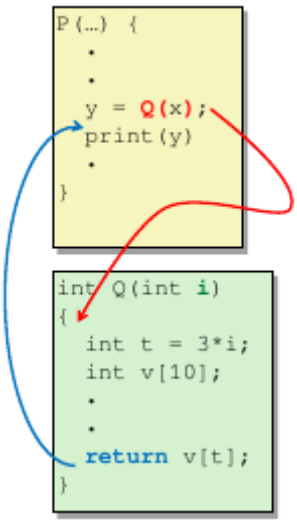
- Control C
  - if-then-else
  - do-while
  - while, for
  - switch
- Control Ensamblador
  - Salto condicional
  - Movimiento condicional
  - Salto indirecto (mediante tablas de saltos)
  - Compilador genera secuencia código p/implementar control más complejo
- Técnicas estándar
  - Bucles convertidos a forma do-while (ó salta-en medio ó copia-test)
  - Sentencias switch grandes usan tablas de saltos
  - Sentencias switch poco densas → árboles decisión (if-elseif-elseif-else)

Tema 2.3: procedimientos

1. Mecanismos

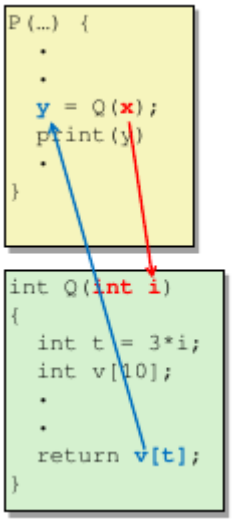
Transferencia de control

- Al principio del código del procedimiento (rojo)
- De vuelta al punto de retorno (azul)



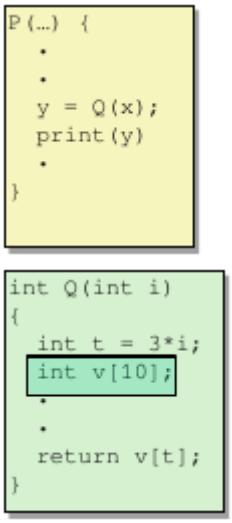
Transferencia de datos

- Argumentos del procedimiento (rojo)
- Valor de retorno (azul)



Gestión de memoria

- Reservar durante ejecución del procedimiento
- Liberar al retornar



Estos mecanismos están todos implementados con instrucciones máquina. La implementación de un proceso concreto usa sólo los mecanismos que este requiera.

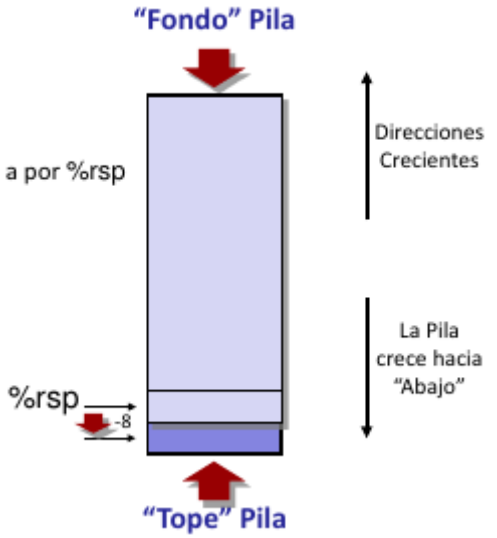
2. Estructura de la pila

Región de memoria gestionada con disciplina de pila, CRECE HASTA POSICIONES INFERIORES. El registro %rsp (puntero de pila) contiene la dirección más baja de la pila (dirección del elemento tope)

Push

```
pushq src
```

1. Capta el operando en src
2. Decrementa %rsp en 8
3. Escribe operando en dirección indicada por %rsp

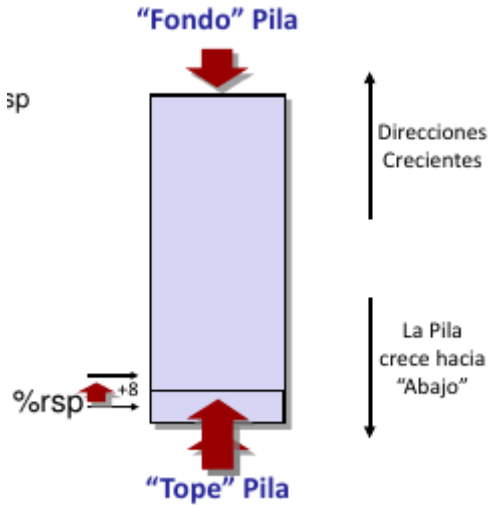


Pop

```
popq dest
```

1. Lee valor de dirección indicada por %rsp
2. Incrementa %rsp en 8

3. Almacena valor en dest



3. Convenciones de llamada

Pasando el control

Código ejemplo

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

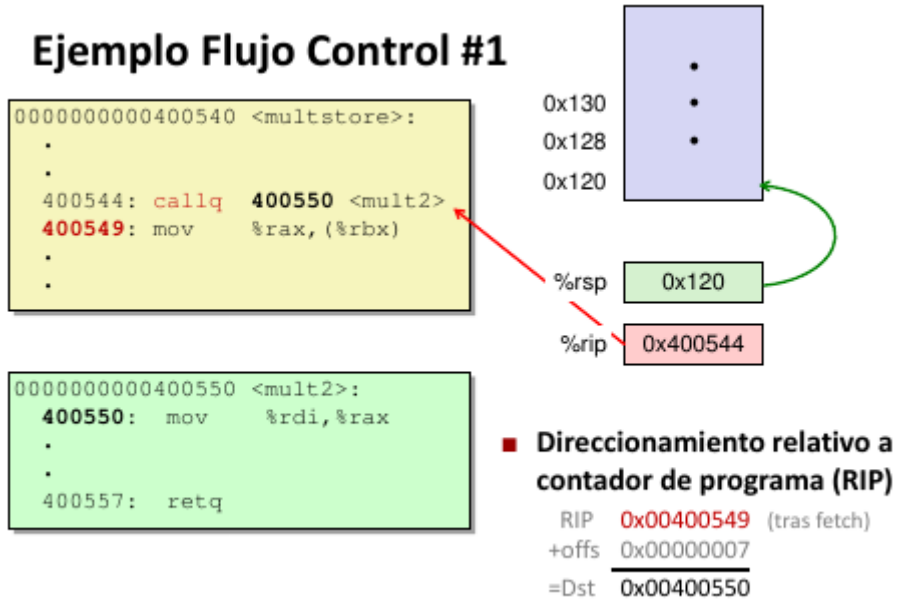
```
0000000000400540 <multstore>:
  400540: push   %rbx              # preservar %rbx
  400541: mov    %rdx,%rbx         # conservar dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # salvar en dest
  40054c: pop    %rbx              # restaurar %rbx
  40054d: retq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

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

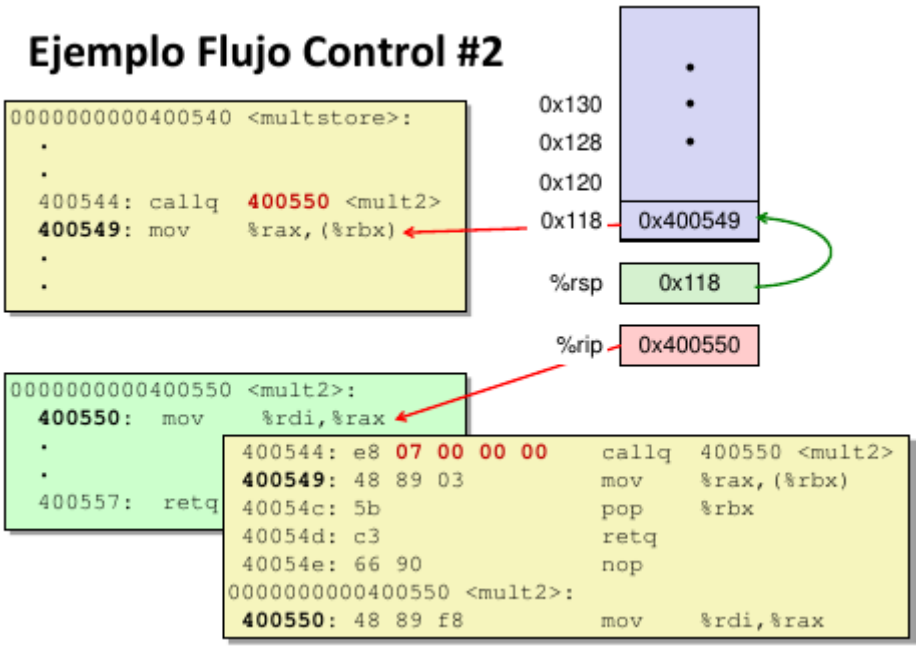
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax      # a
  400553:  imul   %rsi,%rax      # a * b
  400557:  retq                  # retornar
```

Se usa la pila para soportar llamadas y retornos de procedimientos

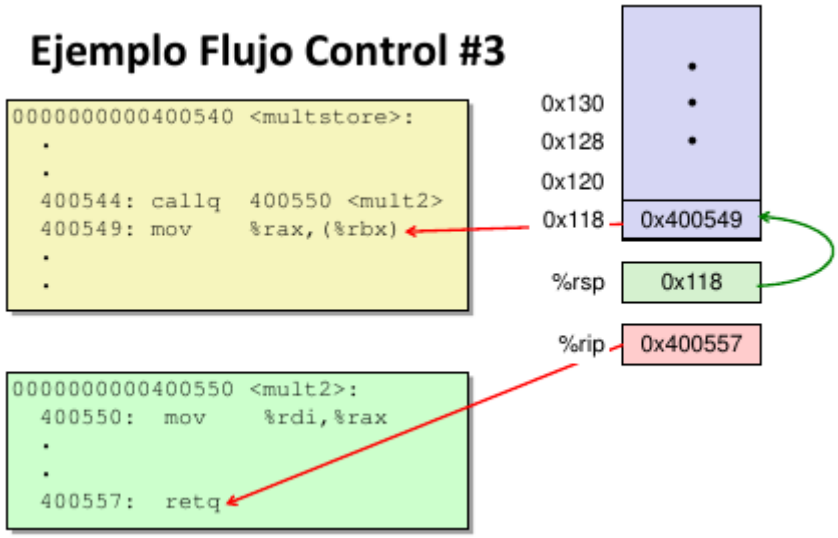
- **Llamada a procedimiento:** `call label`
  - Recuerda (push) la dirección de retorno de la pila
  - Salta a la etiqueta label
  - Codificada con direccionamiento relativo a RIP
- **Dirección de retorno:**
  - Dirección de la siguiente instrucción después de la llamada (call)
    - Ejemplo en desensamblado anterior: 0x400549
- **Retorno de procedimiento:** `ret`
  - Recupera (pop) la dirección de retorno de la pila
  - Salta a dicha dirección



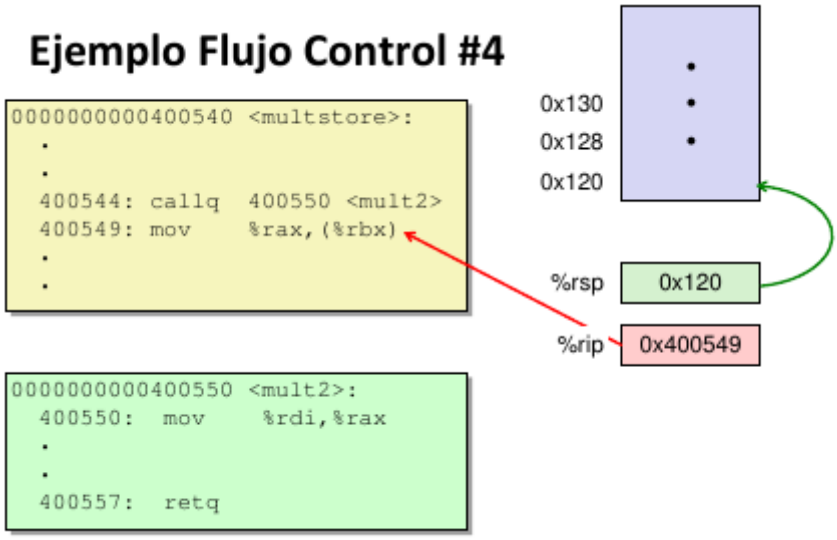
1. Llamada a la función en 0x400544 -> RIP
2. Direccionamiento relativo a RIP
3. RSP apunta a la última dirección de la pila



1. Dirección de la llamada a la pila (RSP se actualiza)
2. RIP da el salto a la label indicada (mult2)



1. RIP ha ido avanzando dentro de la función hasta llegar al retorno
2. Miramos en la pila en dónde nos habíamos quedado y volvemos (pop)



1. Guardamos el resultado devuelto por la función en otro sitio distinto a RAX y continuamos

Pasando los datos

- Registros:
  - Primeros seis argumentos Diane's Silk...
  - Valor de retorno %rax
- Reservamos espacio en la pila cuando se necesite (más de 6 argumentos)

Ejemplo Flujo Datos

```
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
  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
```

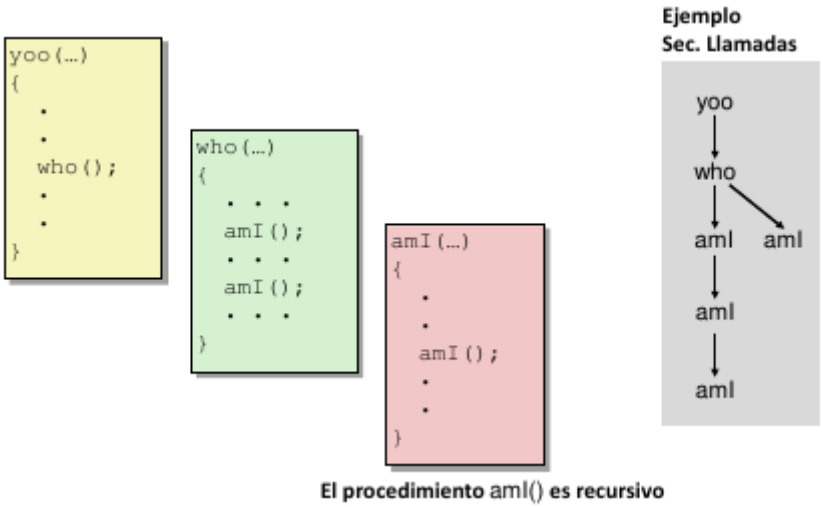
```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

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

Guardar las variables locales en la pila es conveniente para métodos recursivos

Gestionando datos locales

- **Lenguajes basados en pila:** C, Pascal, Java...
  - El código debe ser reentrante
    - Múltiples instanciaciones simultáneas de un mismo procedimiento
  - Se necesita algún lugar donde 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 llama hasta que se retorna)
  - El invocado retorna antes que el invocante
  - La pila se reserva en marcos (estados para una sola instanciación del procedimiento)



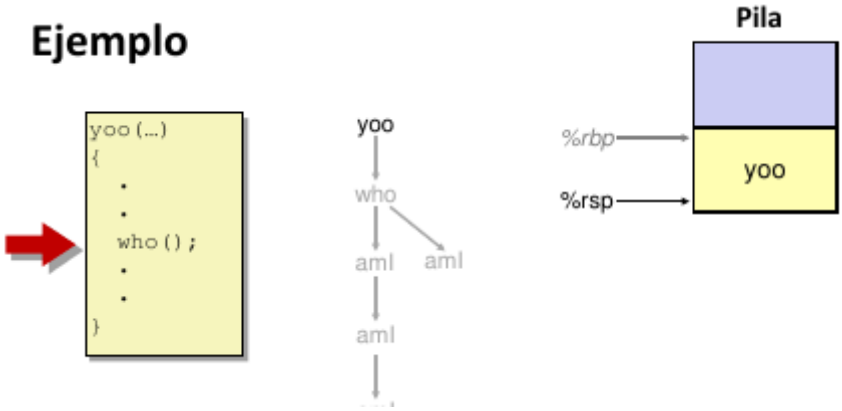
Marcos de pila

Contienen:

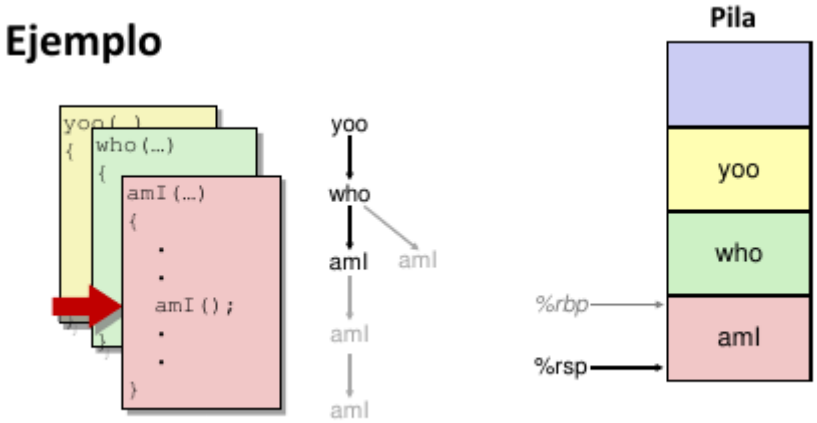
- Información de retorno
- Almacenamiento local (si necesario)
- Espacio temporal (si necesario)

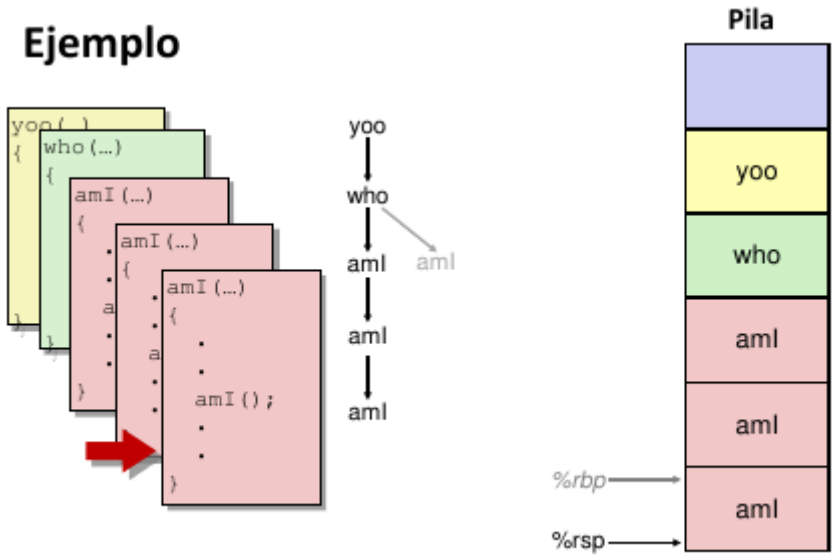
Gestión:

- El espacio se reserva al entrar al procedimiento
    - Código de inicialización
    - Incluye el push de dirección de retorno de la instrucción call
- Se libera el espacio al retornar
    - Código de finalización
    - Incluye el pop de continuar el programa de la instrucción ret

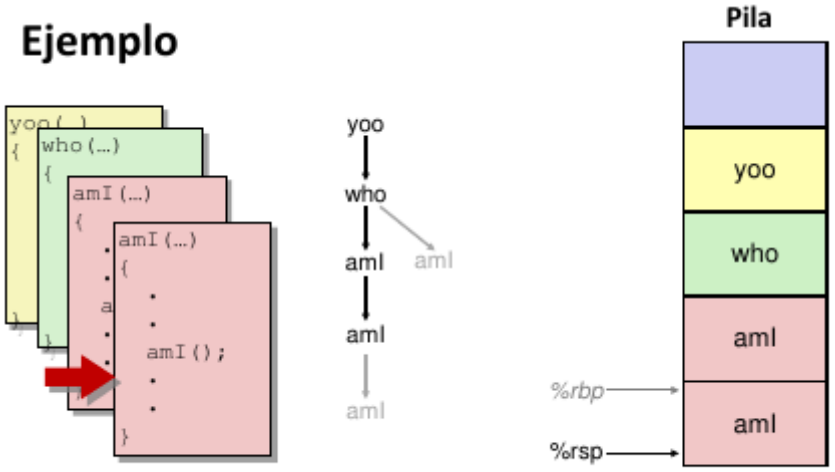


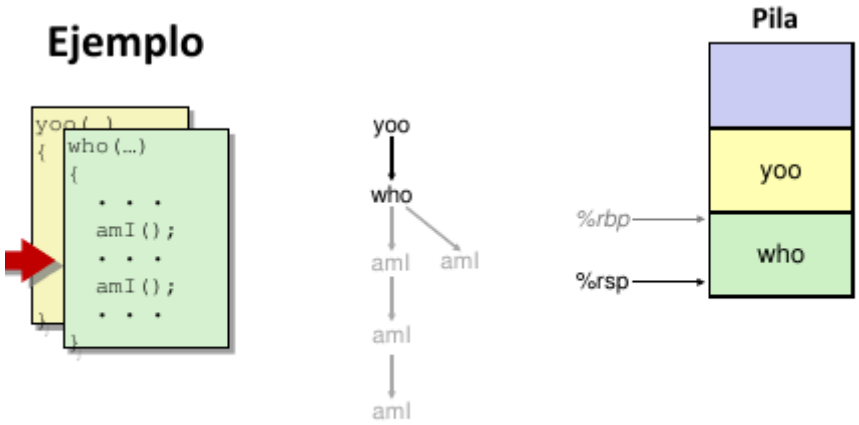
Empezamos en yoo. Vamos bajando por who (que tiene dos llamadas a aml, recursivas)



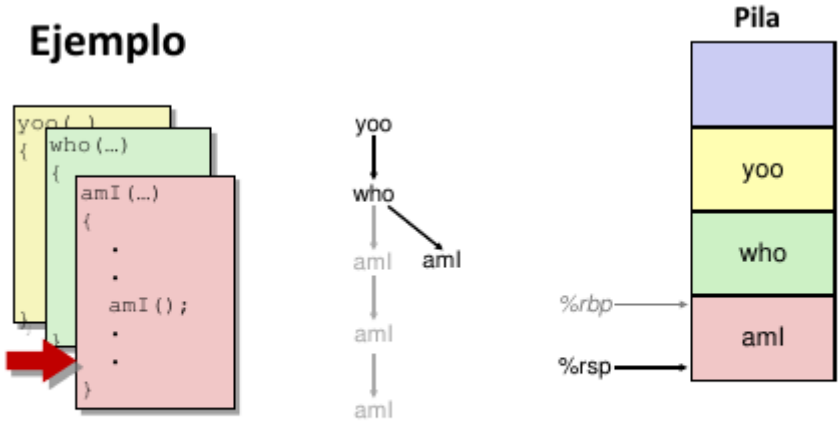


Ha habido tres llamadas recursivas a ami hasta que se ha alcanzado el caso base. Empezamos a volver, haciendo pop en la pila de los marcos de estas funciones

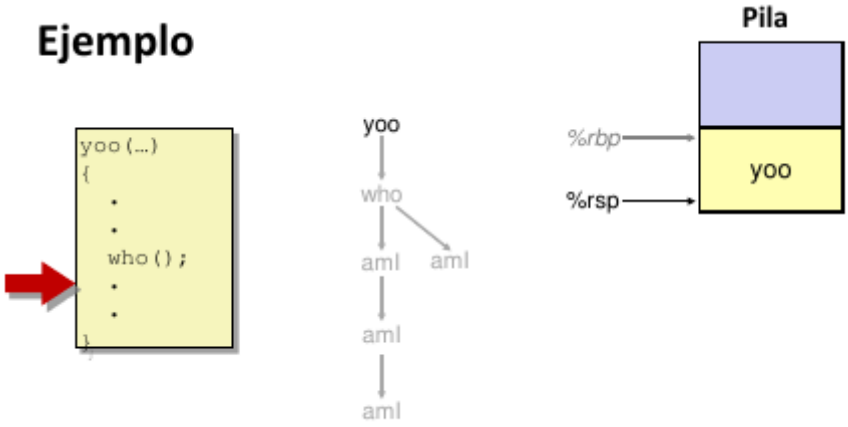




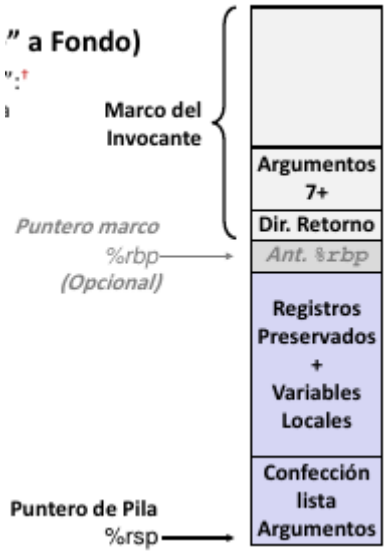
Hemos hecho pop hasta volver a who, donde sigue avanzando el programa hasta llegar a la segunda llamada a ami



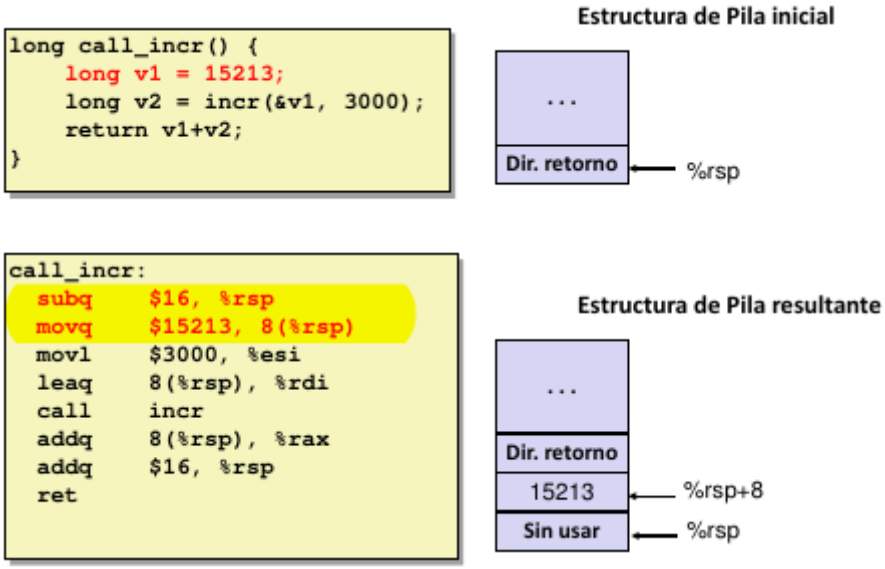
Solo nos requiere una llamada hasta el caso base, hacemos pop de ami y de who para retornar hasta yoo y continuar con la ejecución del programa



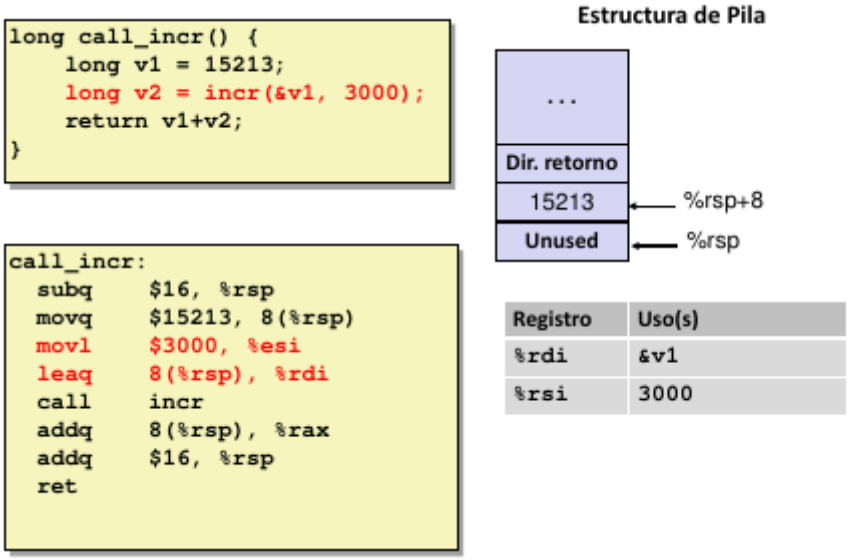
- **Contenidos de marco de pila:** (de tope a fondo)
  - Confección de la lista de argumentos (parámetros mayores que 7 de la función a punto de ser llamada)
  - Variables locales (las que no se pueden mantener en registros)
  - Contexto de registros preservados
  - (Opcional) Antiguo puntero de marco
    - La dirección de retorno pertenece al punto anterior
- **Marco de pila del invocante:**
  - Dirección de retorno
    - Salvada por call
  - Argumentos (más de 7) para la llamada



Ejemplo: llamar a incr



1. Reservamos espacio en la pila (16B, vamos a necesitar dos espacios de 8B)
2. Metemos el primer dato v1 a la pila (local)



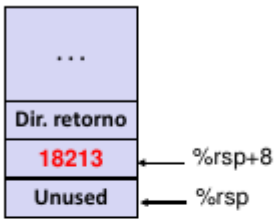
Registro	Uso(s)
%rdi	&v1
%rsi	3000

1.

```
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

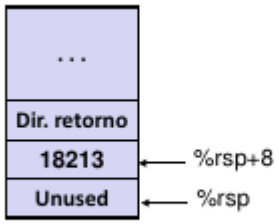


Registro	Uso(s)
%rdi	&v1
%rsi	3000

1.

```
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

1.

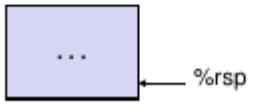
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
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



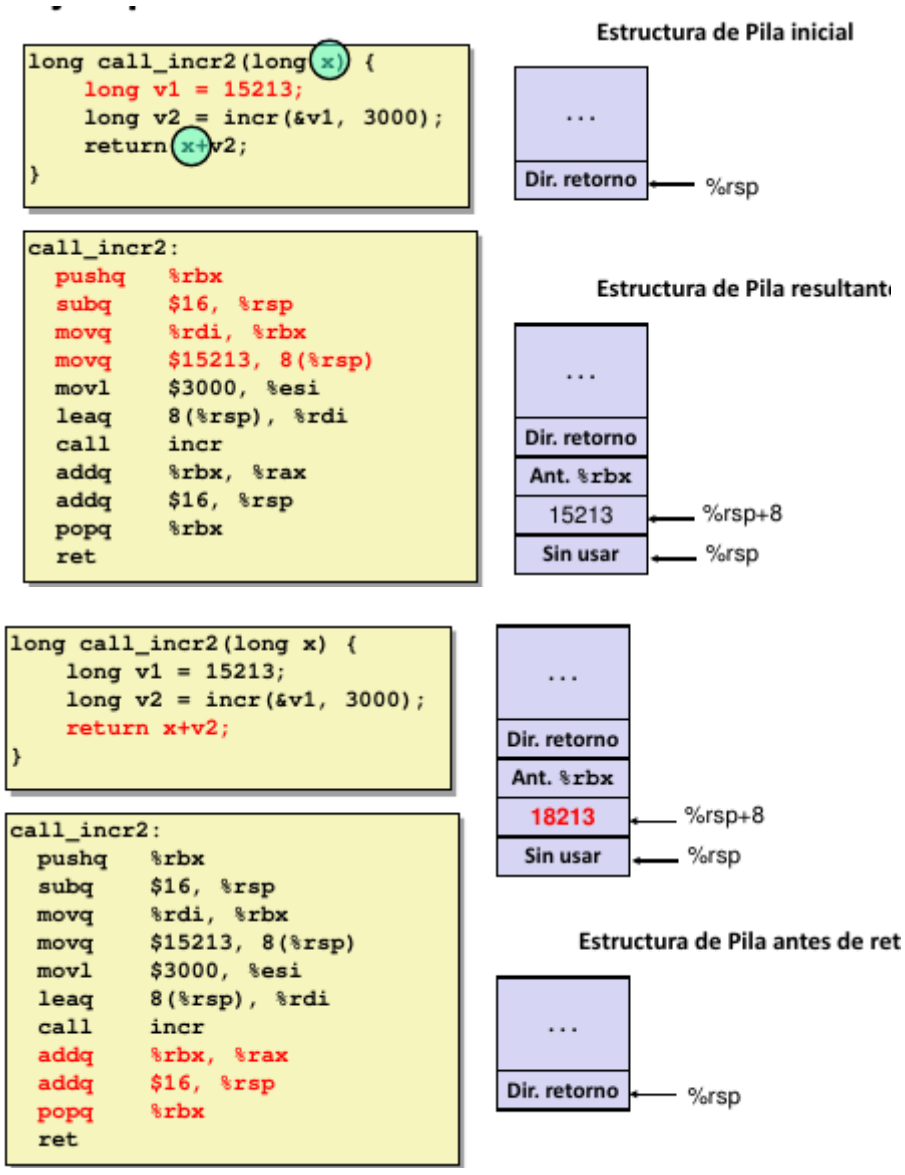
1. Return con RAX

Convenciones de preservación de registros

Cuando el procedimiento yoo llama a who, yoo es el **invocante**, y who es el **invocado**.

¿Podemos usar un registro para almacenamiento temporal dentro de una función? Sí! Pero necesitamos coordinarlo bien y tener cuidado. Si usamos un registro como almacenamiento temporal en la función invocante y en la invocada, tendremos un problema porque la invocada sobreescribirá el contenido. La convención es:

- **Salvainvocante:** el invocante salva los valores temporales en su marco antes de llamar al invocado. Los siguientes registros pueden ser modificados por el programa:
  - %rax - valor de retorno
  - %rdi...%r9 - argumentos
  - %r10, %r11
- **Salvainvocado:** el invocado salva los valores temporales en su marco antes de utilizarlos. Luego los utiliza, hace lo que necesite... y los restaura a como estaban al principio antes del return. Los siguientes registros deben ser preservados y restaurados antes de volver a la función invocante:
  - %rbx, %r12 a %r15, %esi, %edi
  - %rbp - puede que se use como marco de pila
  - %rsp - salvainvocado especial, puntero de pila que se restaura a su valor origina al salir del procedimiento (para volver a la posición anterior de la pila)



4. Ejemplos ilustrativos de recursividad

```
/* 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
```

La condición de terminación es x=0

```
/* 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
```

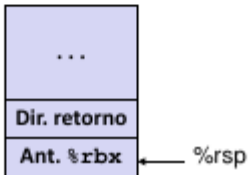
Registro	Uso(s)	Tipo
%rdi	x	Salva-invocante
%rax	Valor de retorno	Salva-invocante

Vemos como se preserva el registro %rbx (salvainvocado) antes de avanzar más en la función:

```
/* 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	Uso(s)	Tipo
%rbx	x & 1	Salva-invocado



Preparar llamada a la función recursiva (salvar los registros de los argumentos [%rdi, salvainvocante])

```
/* 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	Uso(s)	Tipo
%rbx	x & 1	Salva-invocado
%rdi	x >> 1 Argumento recurs.	Salva-invocante

Llamada a la 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
```

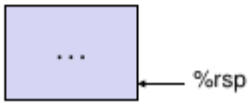
Registro	Uso(s)	Tipo
%rbx	x & 1	Salva-invocado
%rax	Valor de retorno de llamada recursiva	Salva-invocante

Y finalización de la función recursiva (suma del resultado obtenido en %rax, vuelta al bucle)

```
/* 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	Uso(s)	Tipo
%rax	Valor de retorno	Salva-invocante



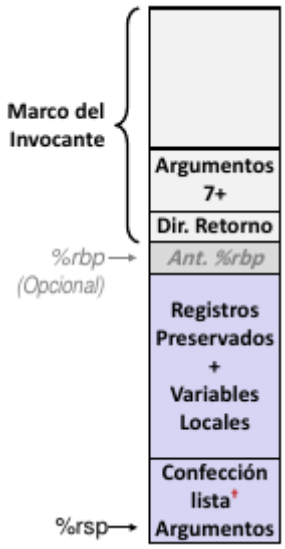
Observaciones sobre recursividad

- Manejada sin especiales consideraciones, ya que…
  - Cada llamada a la función tiene su propio almacenamiento privado gracias a los marcos de pila, que guardan variables locales, preservan los registros, dirección de retorno…
  - Las convenciones de preservación de registros previenen la corrupción de datos entre llamadas, a no ser que el código C lo haga explícitamente (ej. buffer overflow)

    - La disciplina de pila sigue un patrón de llamada/retorno: si P llama a Q, entonces Q retorna antes que P (LIFO, last in first out)
- Lo visto se aplica de igual manera a recursividad mutua (P llama a Q, Q llama a P)

Resumen

- La pila es la estructura de datos correcta para llamada/retorno de procedimientos (LIFO)
- Recursividad con mismas convenciones de llamada que funciones normales
    - Almacenamos valores en el marco de pila local y en registros salvainvocados
    - Argumentos mayores que 7 en el tope de la pila
    - Devolver resultado en %rax
- Punteros son direcciones de valores, global o en pila



Tema 2.4: datos

Tipos de datos básicos

- **Enteros:** almacenados y manipulados en registros enteros de propósito general. Con o sin signo dependerá de las instrucciones usadas

Intel	ASM	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int

- **Coma flotante:** almacenados y manipulados en registros de coma flotante

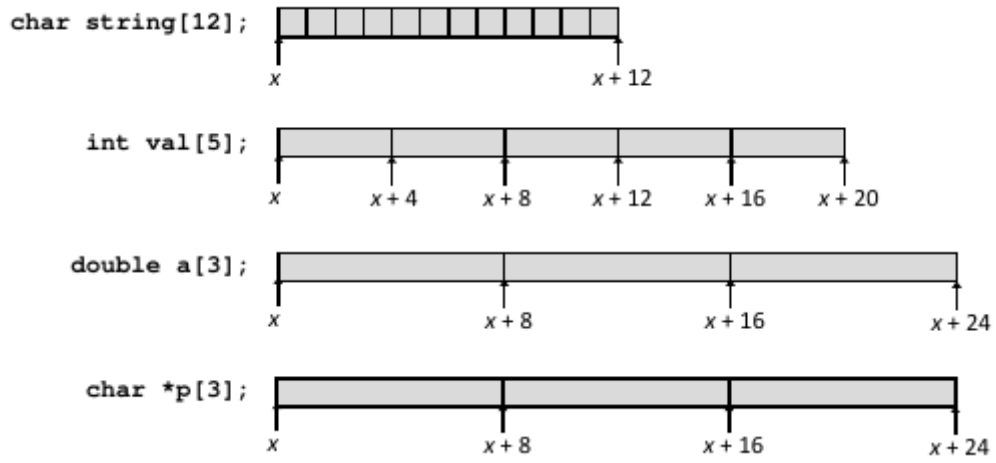
Intel	ASM	Bytes	C
single	s	4	float
double	l	8	double
extended	t	10/12/16	long double

1. Arrays

Unidimensionales

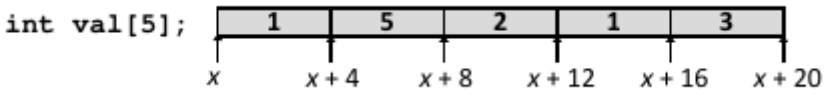
Ubicación de arrays

Principio básico `T A[L]`: array de tipo T llamado A de longitud L. Se reserva la región contigua en memoria de L*sizeof(T) bytes



Acceso a arrays

El identificador A (tipo T*) puede usarse como puntero al elemento 0



Referencia	Tipo	Valor
val[4]	int	1
val	int*	x
val+1	int*	x+4
&val[2]	int*	x+8
val[5]	int	basura
*(val+1)	int	5
val+i	int*	x+4*i

Multidimensionales (anidados)

Multinivel

2. Estructuras

Ubicación

Acceso

Alineamiento

3. Uniones