Lenguaje de máquina: control

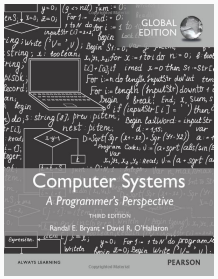
Organización del computador - FIUBA

2.^{do} cuatrimestre de 2023

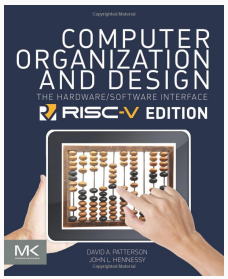
Última modificación: Mon Apr 10 16:58:24 2023 -0300

Créditos

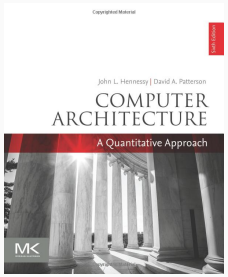
Para armar las presentaciones del curso utilizamos:



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.

El contenido de los slides está basado en las presentaciones de Patricio Moreno y de Organización del Computador I - FCEN.

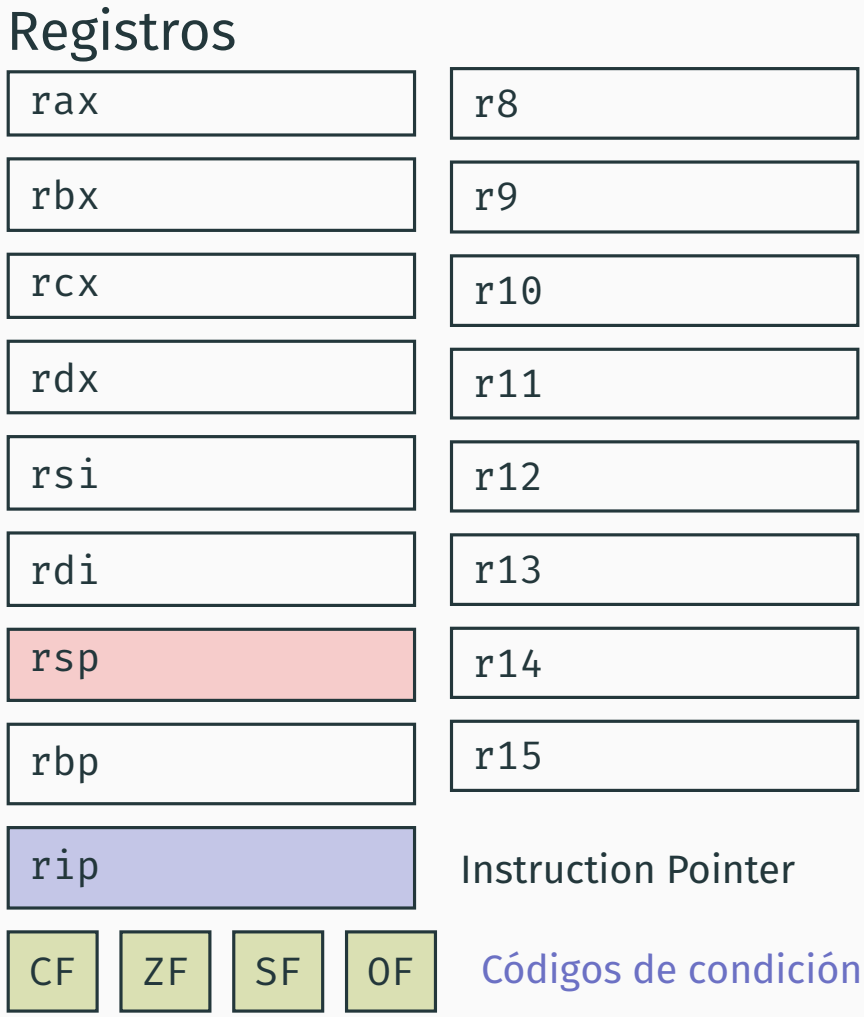
Tabla de contenidos

Tabla de contenidos

Estado parcial del procesador (x86-64)

- Información sobre el proceso que se está ejecutando
  - Datos temporales (`%rax`, ...)
  - Ubicación del *stack* (`%rsp`)
  - Ubicación del *program counter* (`%rip`)
  - Estado de los últimos *test* (**`CF`**, **`ZF`**, **`SF`**, **`OF`**)

`%rsp`: tope del stack



Condiciones: seteo implícito

Registros de un único bit

CF Carry Flag (para `unsigned`)
ZF Zero Flag
SF Sign Flag (para `signed`)
OF Overflow Flag (para `signed`)

Seteo implícito (como efecto secundario) de operaciones aritméticas

Ejemplo: **`addq src, dst`** $\iff$ `t = a + b`

`CF set` si el carry/borrow llega a ser 1 (`unsigned` *overflow*)
`ZF set` si `t == 0`
`SF set` si `t < 0` (`signed`)
`OF set` si hay *overflow* en complemento a dos (`signed`)

```
(a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t >= 0)
```

`leaq` no tiene efectos secundarios

Condiciones: seteo implícito

ZF se activa cuando:

```
0000000000..0000000000
```

Condiciones: seteo implícito

SF se activa cuando:

$$
\begin{array}{rr}
 & x?????????\ldots \\
+ & y?????????\ldots \\
\hline
 & 1?????????\ldots
\end{array}
$$

Condiciones: seteo implícito

CF se activa cuando:

```
      1??????????…
+     1??????????…        Carry
-------------------------
  1   ??????????…

  1   0??????????…
-     1??????????…        Borrow
-------------------------
      1??????????…
```

Condiciones: seteo implícito

OF se activa cuando:

$$\begin{array}{rr} & y?????????\ldots \\ + & y?????????\ldots \\ \hline & z?????????\ldots \end{array} \qquad \begin{array}{rr} & a \\ + & b \\ \hline & t \end{array}$$

```
(a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t >= 0)
```

Condiciones: seteo explícito

Instrucción de comparación

- `cmpq der,izq`
- `cmpq b,a`: `a - b` sin almacenar el resultado

`CF set` si el carry/borrow llega a ser 1 (`unsigned` *overflow*)
`ZF set` si `a == b`
`SF set` si `(a - b) < 0` (`signed`)
`OF set` si hay *overflow* en complemento a dos (`signed`)

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Condiciones: seteo explícito

Instrucción de pruebas (*test*)

- `testq der,izq`
- `testq b,a`: `a & b` sin almacenar el resultado
- Activa los códigos de condición con el resultado de un `AND` binario

```
ZF set  si (a & b) == 0
SF set  si (a & b) < 0
```

Útil para *testear* registros: `testq %rax, %rax`

Condiciones: lectura explícita

Instrucciones `set`

- **`setX dst`**: pone a `0` o `1` el byte menos significativo de **`dst`** en función de uno o más bits de condición.
- No altera los demás bytes

set...	condición	descripción
`sete`	`ZF`	Igual / Cero
`setne`	`~ZF`	No igual / No cero
`sets`	`SF`	Negativo
`setns`	`~SF`	No negativo
`setg`	`~(SF^OF)&~ZF`	Mayor (signado)
`setge`	`~(SF^OF)`	Mayor o igual (signado)
`setl`	`SF^OF`	Menor (signado)
`setle`	`~(SF^OF)|ZF`	Menor o igual (signado)
`seta`	`~CF&~ZF`	Mayor (unsigned)
`setb`	`CF`	Menor (unsigned)

Registros x86-64

rax	al
rbx	bl
rcx	cl
rdx	dl
rsi	sil
rdi	dil
rsp	spl
rbp	bpl

r8	r8b
r9	r9b
r10	r10b
r11	r11b
r12	r12b
r13	r13b
r14	r14b
r15	r15b

Condiciones: lectura explícita

Instrucción `set`

- Modifica únicamente un byte, dejando igual el resto
- Recibe registros de un byte
  - **NO** modifica los bytes superiores
  - Se ponen a cero, típicamente, usando **`movzbl`**
    - Instrucciones de 32 bits ponen a cero los bits superiores

```c
int gt(long a, long b)
{
    return a > b;
}
```

Registro	Uso(s)
`%rdi`	Argumento **a**
`%rsi`	Argumento **b**
`%rax`	Valor de retorno

```
cmpq    %rsi, %rdi   # Compara a e b
setg    %al          # Setea %al si > (~(SF^OF)&~ZF)
movzbl  %al, %eax    # Pone a cero el resto de %rax
ret
```

Tabla de contenidos

Saltos (*branch*)

Instrucciones de salto

- saltan en función de los códigos de condición

j...	condición	descripción
`jmp`	`1`	Incondicional
`je`	`ZF`	Igual / Cero
`jne`	`~ZF`	No igual / No cero
`js`	`SF`	Negativo
`jns`	`~SF`	No negativo
`jg`	`~(SF^OF)&~ZF`	Mayor (signado)
`jge`	`~(SF^OF)`	Mayor o igual (signado)
`jl`	`SF^OF`	Menor (signado)
`jle`	`~(SF^OF)|ZF`	Menor o igual (signado)
`ja`	`~CF&~ZF`	Mayor (unsigned)
`jb`	`CF`	Menor (unsigned)

Saltos (*branch*)

Instrucciones de salto

```
gcc -std=c99 -Wall -pedantic -Og -S absdiff.c
```

```
long absdiff(long x, long y) {
    long result;

    if (x > y)
        result = x - y;
    else
        result = y - x;

    return result;
}
```

```
absdiff:
    cmpq %rsi,%rdi  # cmpq y, x
    jle  .L2        # (x - y) <= 0
    movq %rdi,%rax    # result = x
    subq %rsi,%rax    # result -= y
    ret
.L2:
    movq %rsi,%rax    # result = y
    subq %rdi,%rax    # result -= x
    ret
```

Nuevamente: %rdi ← x,
%rsi ← y,
%rax ← valor de retorno.

Saltos: expresados con goto (!)

- `goto` se utiliza para saltos *absolutos*
- se puede ver un paralelismo entre `goto` y assembly

```
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 Fin;
Else:
    result = y - x;
Fin:
    return result;
}
```

Saltos: expresados con goto (!)

- `goto` se utiliza para saltos *absolutos*
- se puede ver un paralelismo entre `goto` y assembly

```
absdiff:
    cmpq %rsi,%rdi    # x:y
    jle .L2
    movq %rdi,%rax    # result = x
    subq %rsi,%rax    # result -= y
    ret
.L2:
    movq %rsi,%rax    # result = y
    subq %rdi,%rax    # result -= x
    ret
```

```
long absdiff_j(long x, long y) {
    long result;

    int ntest = x <= y;
    if (ntest) goto Else;
    result = x - y;
    goto Fin;
Else:
    result = y - x;
Fin:
    return result;
}
```

Operador condicional: análisis

Código C:

```
val = test ? expr_si : expr_else ;
```

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

Pseudo C/goto

```
    ntest = !test;
    if (ntest) goto Else;
    val = expr_si;
    goto Fin;
Else:
    val = expr_else;
Fin:
    . . .
```

- Crea regiones separadas para el código de cada expresión
- Salta y ejecuta la adecuada

Movimientos condicionales: ¿qué hace el compilador?

- Instrucciones de movimiento condicional
  - Si es soportada, realiza:

    `if (test) dest ← src`
  - Soporte en x86 desde 1995 (más de 20 años ya)
  - `gcc` las usa, cuando es seguro
- Motivación
  - Los saltos son instrucciones disruptivas en el flujo a través de *pipelines* ¿y qué es un pipeline?
  - Los movimientos condicionales no requieren de transferencias de control

- Código C:

```
val = test
    ? expr_si
    : expr_else;
```

- Versión "goto":

```
result = expr_si;
aux = expr_else;
nt = !test;
if (nt) result = aux;
```

no tiene `goto` ⇒ no hay saltos, no hay ramificaciones del código

Movimiento condicional: ejemplo

```c
long absdiff(long x, long y) {
    long result;

    if (x > y)
        result = x - y;
    else
        result = y - x;

    return result;
}
```

Nuevamente:

- %rdi ← x,
- %rsi ← y,
- %rax ← valor de retorno.

```
absdiff:
    movq    %rdi,%rdx   # x
    subq    %rsi,%rdx   # auxiliar = x - y
    movq    %rsi,%rax   # y
    subq    %rdi,%rax   # resultado = y - x
    cmpq    %rsi,%rdi  # compara x e y
    cmovg   %rdx,%rax   # if (x > y) resultado = auxiliar
    ret
```

Movimiento condicional: casos malos

Cálculos costosos

```
valor = test(x) ? hard1(x) : hard2(x);
```

- Ambos cálculos deben ser simples

Mala performance

Cálculos inseguros

```
valor = p ? *p : 0;
```

Inseguro

- Puede tener efectos indeseados

Cálculos con efectos secundarios

```
valor = x > 0 ? x *= 7 : x += 3;
```

Ilegales

- Deben NO tener efectos secundarios

Ejercicio

`cmpq b,a`: **`a-b`** sin almacenar el resultado

CF set si el carry/borrow llega a ser 1 (**unsigned** *overflow*)

ZF set si `a == b`

SF set si `(a - b) < 0` (**signed**)

OF set si hay *overflow* en complemento a dos (**signed**)

set...	condición	descripción
sete	ZF	Igual / Cero
setne	∼ZF	No igual / No cero
sets	SF	Negativo
setns	∼SF	No negativo
setg	∼(SF^OF)&∼ZF	Mayor (signado)
setge	∼(SF^OF)	Mayor o igual (signado)
setl	SF^OF	Menor (signado)
setle	∼(SF^OF)|ZF	Menor o igual (signado)
seta	∼CF&∼ZF	Mayor (unsigned)
setb	CF	Menor (unsigned)

```
xorq    %rax,%rax
subq    $1,%rax
cmpq    $2,%rax
setl    %al
movzbl  %al,%eax
```

%rax	SF	CF	OF	ZF

Ejercicio

`cmpq b,a`: **`a-b`** sin almacenar el resultado

CF set si el carry/borrow llega a ser 1 (**unsigned** *overflow*)

ZF set si `a == b`

SF set si `(a - b) < 0` (**signed**)

OF set si hay *overflow* en complemento a dos (**signed**)

set...	condición	descripción
`sete`	`ZF`	Igual / Cero
`setne`	`~ZF`	No igual / No cero
`sets`	`SF`	Negativo
`setns`	`~SF`	No negativo
`setg`	`~(SF^OF)&~ZF`	Mayor (signado)
`setge`	`~(SF^OF)`	Mayor o igual (signado)
`setl`	`SF^OF`	Menor (signado)
`setle`	`~(SF^OF)|ZF`	Menor o igual (signado)
`seta`	`~CF&~ZF`	Mayor (unsigned)
`setb`	`CF`	Menor (unsigned)

		%rax	SF	CF	OF	ZF
`xorq`	`%rax,%rax`	`0x0000 0000 0000 0000`	0	0	0	1
`subq`	`$1,%rax`	`0xFFFF FFFF FFFF FFFF`	1	1	0	0
`cmpq`	`$2,%rax`	`0xFFFF FFFF FFFF FFFF`	1	0	0	0
`setl`	`%al`	`0xFFFF FFFF FFFF FF01`	1	0	0	0
`movzbl`	`%al,%eax`	`0x0000 0000 0000 0001`	1	0	0	0

Tabla de contenidos

Ciclo: do-while

```
long pcount_do (unsigned long x)
     {
    long resultado = 0;

    do {
        resultado += x & 0x1;
        x >>= 1;
    } while (x) ;

    return resultado;
}
```

```
long pcount_goto (unsigned long
    x) {
    long resultado = 0;

loop:
    resultado += x & 0x1;
    x >>= 1;
    if (x) goto loop;

    return resultado;
}
```

- Cuenta la cantidad de unos (1s) en el argumento x (*popcount*)
- Usa un salto condicional para seguir iterando o salir del ciclo

Compilación de do-while

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

Registro	Uso
%rdi	x
%rax	resultado

```
pcount_goto:
    movl    $0, %eax    # resultado = 0
.L2:                    # loop:
    movq    %rdi, %rdx
    andl    $1, %edx    # temp = x & 0x1
    addq    %rdx, %rax  # resultado += temp
    shrq    %rdi        # x >>= 1
    jne     .L2         # if (x) goto loop;
    rep ret             # https://repzret.org/p/repzret/
```

“Traducción” general #1 `do-while`

(pseudo)código C

```
do {
    cuerpo
} while (test);
```

(pseudo)“código” C

```
loop:
    cuerpo
if (test)
    goto loop;
```

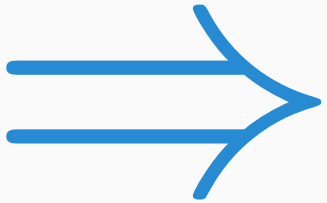
- cuerpo: {
  instrucción_1;
  instrucción_2;
  ⋮
  instrucción_n;
  }

“Traducción” general #1 del `while`

- “traducción” *jump-to-middle* (`-Og`)

(pseudo)código C

```
while (test)
    cuerpo
```



(pseudo)“código” C

```
goto prueba;
loop:
    cuerpo
prueba:
    if (test)
        goto loop;
fin:
```

Ejemplo

código C

```
long pcount_while (unsigned
    long x) {
    long resultado = 0;

    while (x) {
        resultado += x & 0x1
            ;
        x >>= 1;
    }

    return resultado;
}
```

jump-to-middle

```
long pcount_goto_jtm (
    unsigned long x) {
    long resultado = 0;

    goto test;
loop:
    resultado += x & 0x1;
    x >>= 1;
test:
    if (x) goto loop;

    return resultado;
}
```

Ejemplo

código assembly

```
    movl    $0, %eax
    jmp     .L2
.L3:
    movq    %rdi, %rdx
    andl    $1, %edx
    addq    %rdx, %rax
    shrq    %rdi
.L2:
    testq   %rdi, %rdi
    jne     .L3
    rep ret
```

jump-to-middle

```
long pcount_goto_jtm (
    unsigned long x) {
    long resultado = 0;

    goto test;
loop:
    resultado += x & 0x1;
    x >>= 1;
test:
    if (x) goto loop;

    return resultado;
}
```

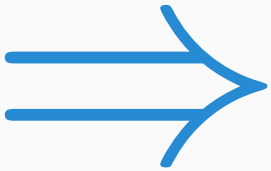
“Traducción” general #2 del `while`

while

```
while (test)
    cuerpo
```

“do-while”

```
    if (!test)
        goto fin;
    do
        cuerpo
    while (test) ;
fin:
```



(pseudo)“código” C

```
    if (!test)
        goto fin;
loop:
    cuerpo
    if (test)
        goto loop;
fin:
```

- Conversión a `do-while`
- Se obtiene con `-O1`

Ejemplo

código C

```
long pcount_while (unsigned
    long x) {
    long resultado = 0;

    while (x) {
        resultado += x & 0x1
            ;
        x >>= 1;
    }

    return resultado;
}
```

método #2

```
long pcount_goto_dw (
    unsigned long x) {
    long resultado = 0;

    if (!x) goto fin;

loop:
    resultado += x & 0x1;
    x >>= 1;
    if (x) goto loop;

fin:
    return resultado;
}
```

- Una prueba antes del “`do-while`”

Ejemplo

código assembly

```
pcount_while:
    testq   %rdi, %rdi
    je      .L4
    movl    $0, %eax
.L3:
    movq    %rdi, %rdx
    andl    $1, %edx
    addq    %rdx, %rax
    shrq    %rdi
    jne     .L3
    rep ret
.L4:
    movl    $0, %eax
    ret
```

método #2

```
long pcount_goto_dw (
    unsigned long x) {
    long resultado = 0;

    if (!x) goto fin;

loop:
    resultado += x & 0x1;
    x >>= 1;
    if (x) goto loop;

fin:
    return resultado;
}
```

Ciclos: for

Estructura

```
for (init; test; update)
    cuerpo
```

```c
#include <stdlib.h>
#define WSIZE 8*sizeof(int)
long pcount_for (unsigned long x) {
    size_t i;
    long resultado = 0;

    for (i = 0; i < WSIZE; i++) {
        unsigned bit =
            (x >> i) & 0x1;
        resultado += bit;
    }

    return resultado;
}
```

init

```
i = 0
```

test

```
i < WSIZE
```

update

```
i++
```

cuerpo

```
{
    unsigned bit =
        (x >> i) & 0x1;
    resultado += bit;
}
```

Ciclos: de un `for` a un `while`

Versión for

```
for (init; test; update)
    cuerpo
```

Versión while

```
init;
while (test) {
    cuerpo
    update;
}
```

Ciclos: `for` a `while`

init

```
i = 0
```

test

```
i < WSIZE
```

update

```
i++
```

cuerpo

```
{
  unsigned bit =
      (x >> i) & 0x1;
  resultado += bit;
}
```

```
#include <stdlib.h>
#define WSIZE 8*sizeof(int)
long pcount_for_while (unsigned
    long x) {
    size_t i;
    long resultado = 0;

    i = 0;
    while(i < WSIZE) {
        unsigned bit =
            (x >> i) & 0x1;
        resultado += bit;
        i++;
    }

    return resultado;
}
```

Ciclos: `for` a `do-while`

código C

```c
#include <stdlib.h>
#define WSIZE 8*sizeof(int)
long pcount_for (unsigned long x) {
    size_t i;
    long resultado = 0;

    for (i = 0; i < WSIZE; i++) {
        unsigned bit =
            (x >> i) & 0x1;
        resultado += bit;
    }

    return resultado;
}
```

reversionado

```c
long pcount_for_goto_dw (unsigned
    long x) {
    size_t i;
    long resultado = 0;
    i = 0;   // init
    if (!(i < WSIZE)) goto fin;
loop:
  {
    unsigned bit =
        (x >> i) & 0x1; // cuerpo
    resultado += bit;
  }
    i++;     // update
    if ((i < WSIZE)) goto loop;
                          // test
fin:
    return resultado;
}
```

Ciclos: `for` a `do-while`

código C

```
#include <stdlib.h>
#define WSIZE 8*sizeof(int)
long pcount_for (unsigned long x) {
    size_t i;
    long resultado = 0;

    for (i = 0; i < WSIZE; i++) {
        unsigned bit =
            (x >> i) & 0x1;
        resultado += bit;
    }

    return resultado;
}
```

Se puede eliminar la primera prueba

reversionado

```
long pcount_for_goto_dw (unsigned
    long x) {
    size_t i;
    long resultado = 0;
    i = 0;   // init
    if (!(i < WSIZE)) goto fin;
loop:
  {
    unsigned bit =
        (x >> i) & 0x1; // cuerpo
    resultado += bit;
  }
    i++;     // update
    if ((i < WSIZE)) goto loop;
                          // test
fin:
    return resultado;
}
```

Tabla de contenidos

`switch`: ejemplo

```
long switch_fun (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;
}
```

Particularidades del `switch`

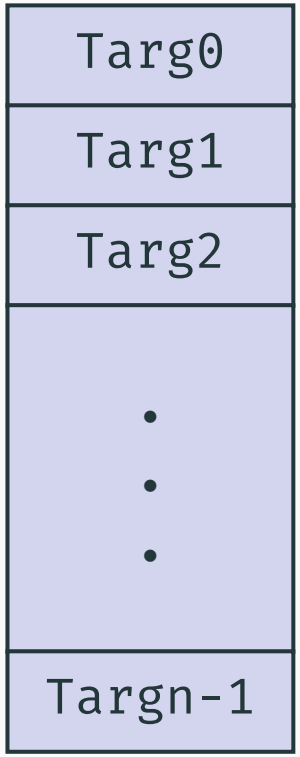
- *Etiquetas múltiples:*
  - En los casos 5 y 6
- *Fall Through:*
  - En el caso 2
- *Casos faltantes:*
  - El caso 4

`switch`: *jump table*

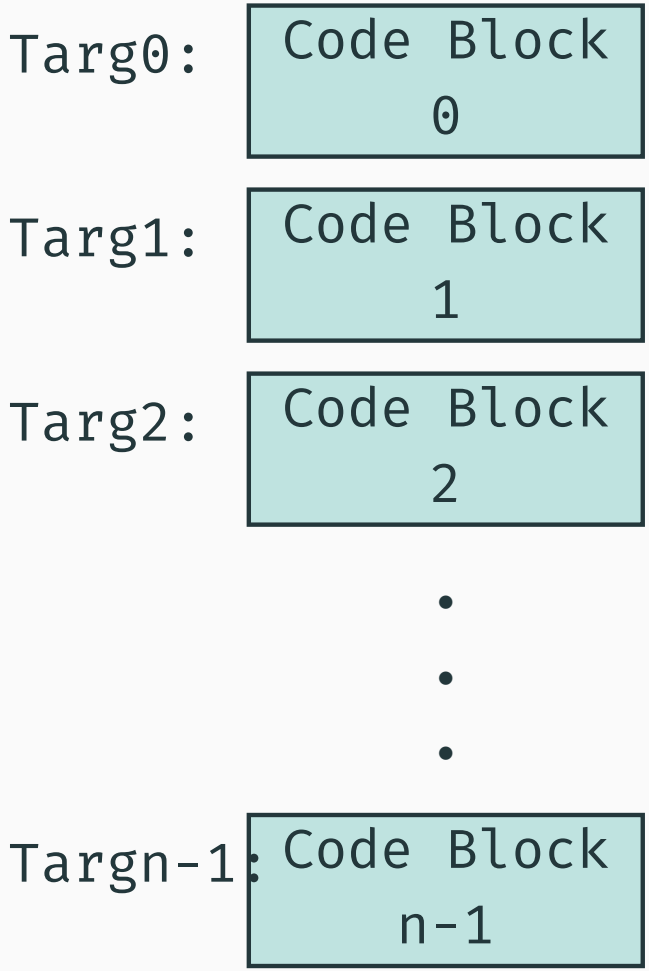
switch en C

```
switch (x) {
    case val_0:
        bloque 0
    case val_1:
        bloque 1
    .
    .
    .
    case val_n-1:
        bloque n-1
}
```

jump table



jump targets



versión **goto**:

```
goto *jtab[x];
```

switch: ejemplo

código C

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

código assembly

Inicialización

```
switch_fun:
    movq    %rdx, %rcx
    cmpq    $6, %rdi
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

Nuevamente:

- %rdi ← x,
- %rsi ← y,
- %rdx ← z,
- %rax ← valor de retorno.

¿y w?

switch: ejemplo

código C

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

código assembly

Inicialización

```
switch_fun:
    movq    %rdx, %rcx
    cmpq    $6, %rdi            # x:6
    ja      .L8                 # default
    jmp     *.L4(,%rdi,8)       # salto
```

jump table

```
  .section  .rodata
  .align 8
  .align 4
.L4:
  .quad   .L8   # x = 0
  .quad   .L3   # x = 1
  .quad   .L5   # x = 2
  .quad   .L9   # x = 3
  .quad   .L8   # x = 4
  .quad   .L7   # x = 5
  .quad   .L7   # x = 6
```

`switch`: assembly

- Estructura de la tabla
  - Cada destino requiere 8 bytes
  - Dirección base en `.L4`
- Saltos
  - **Directo:** `jmp .L8`
  - La dirección es la etiqueta `.L8`
  - **Indirecto:** `jmp *.L4(,%rdi,8)`
  - Comienzo de la tabla: `.L4`
  - Se debe escalar por un factor de 8
  - Obtener el destino de la dirección efectiva `.L4 + x*8`
    - Únicamente para $0 \leq x \leq 6$

jump table

```
  .section  .rodata
  .align 8
  .align 4
.L4:
  .quad  .L8  # x = 0
  .quad  .L3  # x = 1
  .quad  .L5  # x = 2
  .quad  .L9  # x = 3
  .quad  .L8  # x = 4
  .quad  .L7  # x = 5
  .quad  .L7  # x = 6
```

`switch`: relaciones

jump table

```
    .section    .rodata
    .align 8
    .align 4
.L4:
    .quad    .L8    # x = 0
    .quad    .L3    # x = 1
    .quad    .L5    # x = 2
    .quad    .L9    # x = 3
    .quad    .L8    # x = 4
    .quad    .L7    # x = 5
    .quad    .L7    # x = 6
```

```
    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;
    }
}
```

Code blocks: x == 1

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

```
.L3:
  movq    %rsi, %rax
  imulq   %rdx, %rax
  ret
```

Code blocks: *Fall-Through*

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

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```

Code blocks: x == 2, 3

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

```
.L5:                    # case 2
  movq  %rsi, %rax
  cqto                  # extender
                        # con signo
  idivq %rcx            # y/z
  jmp   .L6             # jmp a merge
.L9:                    # case 3
  movl  $1, %eax        # w = 1
.L6:                    # merge:
  addq  %rcx, %rax      # w += z
  ret
```

Code blocks: x == 5, 6, default

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

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

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`