

Lenguaje de máquina: control

95.57/75.03 Organización del computador

Docentes: Patricio Moreno y Adeodato Simó

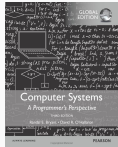
1.^{er} cuatrimestre de 2020

Última modificación: Sun Apr 12 03:52:06 2020 -0300

Facultad de Ingeniería (UBA)

Créditos

Para armar las presentaciones del curso nos basamos en:



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



D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2018.



J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2019.

Tabla de contenidos

1. Condition codes
2. Saltos condicionales
3. Ciclos
4. Switch

Tabla de contenidos

1. Condition codes
2. Saltos condicionales
3. Ciclos
4. Switch

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

Registros

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15
%rip	Instruction Pointer
CF	
ZF	Códigos de condición
SF	
OF	

Condiciones: seteo implícito

Registros de un único bit

CF	Carry Flag (para unsigned)	SF	Sign Flag (para signed)
ZF	Zero Flag	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:

000000000000...000000000000

Condiciones: seteo implícito

SF se activa cuando:

$$\begin{array}{r} x??????????... \\ + \quad y??????????... \\ \hline 1??????????... \end{array}$$

Condiciones: seteo implícito

CF se activa cuando:

$$\begin{array}{r}
 1???????????... \\
 + \quad 1???????????... \\
 \hline
 1 \quad ????????????...
 \end{array}
 \quad \text{Carry}$$

$$\begin{array}{r}
 1 \quad 0???????????... \\
 - \quad 1???????????... \\
 \hline
 1???????????...
 \end{array}
 \quad \text{Borrow}$$

Condiciones: seteo implícito

OF se activa cuando:

$$\begin{array}{r}
 y?????????... \\
 + \quad y?????????... \\
 \hline
 \textcolor{red}{z}?????????...
 \end{array}
 \qquad
 \begin{array}{r}
 a \\
 + \quad b \\
 \hline
 t
 \end{array}$$

$$(a > 0 \ \&\& \ b > 0 \ \&\& \ t < 0) \ || \ (a < 0 \ \&\& \ b < 0 \ \&\& \ t \geq 0)$$

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	\sim ZF	No igual / No cero
sets	SF	Negativo
setns	\sim SF	No negativo
setg	\sim (SF \wedge OF) \wedge \sim ZF	Mayor (signado)
setge	\sim (SF \wedge OF)	Mayor o igual (signado)
setl	SF \wedge OF	Menor (signado)
setle	\sim (SF \wedge OF) ZF	Menor o igual (signado)
seta	\sim CF \wedge \sim 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

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

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

```

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

Tabla de contenidos

1. Condition codes
2. Saltos condicionales
3. Ciclos
4. Switch

Saltos (*branch*)

Instrucciones de salto

- saltan en función de los códigos de condición

j...	condición	descripción
jmp	1	Incondicional
je	ZF	Igual / Cero
jne	\sim ZF	No igual / No cero
js	SF	Negativo
jns	\sim SF	No negativo
jg	\sim (SF \wedge OF) $\&$ \sim ZF	Mayor (signado)
jge	\sim (SF \wedge OF)	Mayor o igual (signado)
jl	SF \wedge OF	Menor (signado)
jle	\sim (SF \wedge OF) ZF	Menor o igual (signado)
ja	\sim CF $\&$ \sim 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: $\%rdi \leftarrow x$,
 $\%rsi \leftarrow y$,
 $\%rax \leftarrow$ valor de retorno.

Salto: 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;  
}
```

Salto: 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:

```
1  val = test ? expr_si : expr_else ;  
2  
3      val = x > y ? x - y : y - x;
```

Pseudo C/goto

```
1      ntest = !test;  
2      if (ntest) goto Else;  
3      val = expr_si;  
4      goto Fin;  
5  Else:  
6      val = expr_else;  
7  Fin:  
8      . . .
```

- 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:
 $\text{if (test) dest} \leftarrow \text{src}$
- 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:

```
1  val = test
2      ? expr_si
3      : expr_else;
```

■ Versión “goto”:

```
1  result = expr_si;
2  aux = expr_else;
3  nt = !test;
4  if (nt) result = aux;
```

no tiene goto \Rightarrow no hay saltos, no hay ramificaciones del código

Movimiento condicional: ejemplo

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

Nuevamente:

`%rdi` \leftarrow `x`,

`%rsi` \leftarrow `y`,

`%rax` \leftarrow 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

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

- Ambos cálculos deben ser simples

Mala performance

Cálculos inseguros

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

- Puede tener efectos indeseados

Inseguro

Cálculos con efectos secundarios

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

llegales

- 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
<code>sete</code>	ZF	Igual / Cero
<code>setne</code>	$\sim ZF$	No igual / No cero
<code>sets</code>	SF	Negativo
<code>setns</code>	$\sim SF$	No negativo
<code>setg</code>	$\sim (SF \wedge OF) \wedge \sim ZF$	Mayor (signado)
<code>setge</code>	$\sim (SF \wedge OF)$	Mayor o igual (signado)
<code>setl</code>	$SF \wedge OF$	Menor (signado)
<code>setle</code>	$\sim (SF \wedge OF) \vee ZF$	Menor o igual (signado)
<code>seta</code>	$\sim CF \wedge \sim ZF$	Mayor (unsigned)
<code>setb</code>	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
<code>sete</code>	ZF	Igual / Cero
<code>setne</code>	\sim ZF	No igual / No cero
<code>sets</code>	SF	Negativo
<code>setns</code>	\sim SF	No negativo
<code>setg</code>	$\sim(SF \wedge OF) \wedge \sim ZF$	Mayor (signado)
<code>setge</code>	$\sim(SF \wedge OF)$	Mayor o igual (signado)
<code>setl</code>	$SF \wedge OF$	Menor (signado)
<code>setle</code>	$\sim(SF \wedge OF) \vee ZF$	Menor o igual (signado)
<code>seta</code>	$\sim CF \wedge \sim ZF$	Mayor (unsigned)
<code>setb</code>	CF	Menor (unsigned)

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

Tabla de contenidos

1. Condition codes
2. Saltos condicionales
3. Ciclos
4. Switch

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

```

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

```

“Traducción” general #1 **do-while**

(pseudo)código C

```
1  do {  
2      cuerpo  
3  } while (test);
```

(pseudo)“código” C

```
1  loop:  
2      cuerpo  
3  if (test)  
4      goto loop;
```

■ **cuerpo:** {
 instrucción₁;
 instrucción₂;
 :
 instrucción_n;
}

“Traducción” general #1 del **while**

- “traducción” *jump-to-middle* (-Og)

(pseudo)código C

```
1 while (test)
2     cuerpo
```



(pseudo)“código” C

```
1 goto prueba;
2 loop:
3     cuerpo
4 prueba:
5     if (test)
6         goto loop;
7 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

```
1      movl    $0, %eax
2      jmp     .L2
3  .L3:
4      movq    %rdi, %rdx
5      andl    $1, %edx
6      addq    %rdx, %rax
7      shrq    %rdi
8  .L2:
9      testq   %rdi, %rdi
10     jne     .L3
11     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


```
1  while (test)
2      cuerpo
```



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

“do-while”

```
1      if (!test)
2          goto fin;
3      do
4          cuerpo
5          while (test) ;
6  fin:
```



(pseudo)“código” C

```
1      if (!test)
2          goto fin;
3  loop:
4      cuerpo
5      if (test)
6          goto loop;
7  fin:
```

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

```
1  pcount_while:
2      testq    %rdi, %rdi
3      je       .L4
4      movl     $0, %eax
5  .L3:
6      movq     %rdi, %rdx
7      andl     $1, %edx
8      addq     %rdx, %rax
9      shrq     %rdi
10     jne       .L3
11     rep ret
12  .L4:
13     movl     $0, %eax
14     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

```

1  for (init; test; update)
2      cuerpo

```

```

#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

```

1  {
2      unsigned bit =
3          (x >> i) & 0x1;
4      resultado += bit;
5  }

```

Ciclos: de un **for** a un **while**

— Versión for —

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



— Versión while —

```
1  init;  
2  while (test) {  
3      cuerpo  
4      update;  
5  }
```

Ciclos: **for** a **while**

init

i = 0

test

i < WSIZE

update

i++

cuerpo

```
1  {  
2      unsigned bit =  
3          (x >> i) & 0x1;  
4      resultado += bit;  
5  }
```

```
#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

```
#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

```
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

1. Condition codes
2. Saltos condicionales
3. Ciclos
4. Switch

switch: ejemplo

```
1 long switch_fun (long x, long y, long z) {  
2     long w = 1;  
3     switch (x) {  
4         case 1:  
5             w = y*z;  
6             break;  
7         case 2:  
8             w = y/z;  
9             /* Fall Through */  
10        case 3:  
11            w += z;  
12            break;  
13        case 5:  
14        case 6:  
15            w -= z;  
16            break;  
17        default:  
18            w = 2;  
19    }  
20    return w;  
21 }
```

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

```

1  switch (x) {
2      case val_0:
3          bloque 0
4      case val_1:
5          bloque 1
6      .
7      .
8      .
9      case val_n-1:
10         bloque n-1
11 }

```

jump table

Targ0
Targ1
Targ2
•
•
•
Targn-1

jump targets

Targ0:	Code Block 0
Targ1:	Code Block 1
Targ2:	Code Block 2
	•
	•
	•
Targn-1:	Code Block n-1

versión goto: goto *jtab[x];

switch: ejemplo

código C

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

código assembly

—— Inicialización ——

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

Nuevamente:

%rdi \leftarrow x,

%rsi \leftarrow y,

%rdx \leftarrow z,

%rax \leftarrow 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

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

jump table

```
1      .section .rodata
2      .align 8
3      .align 4
4      .L4:
5      .quad   .L8   # x = 0
6      .quad   .L3   # x = 1
7      .quad   .L5   # x = 2
8      .quad   .L9   # x = 3
9      .quad   .L8   # x = 4
10     .quad   .L7   # x = 5
11     .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

```
1      .section .rodata
2      .align 8
3      .align 4
4      .L4:
5      .quad .L8  # x = 0
6      .quad .L3  # x = 1
7      .quad .L5  # x = 2
8      .quad .L9  # x = 3
9      .quad .L8  # x = 4
10     .quad .L7  # x = 5
11     .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;  
    . . .  
}
```

```
1  .L3:  
2      movq    %rsi, %rax  
3      imulq   %rdx, %rax  
4      ret
```

Code blocks: *Fall-Through*

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

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

```
1
2 case 3:
3     w = 1;
4 merge:
5     w += z;
```

Code blocks: $x == 2, 3$

```

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

```

```

1  .L5:                                # case 2
2      movq    %rsi, %rax
3      cqto                                # extender
4                                          # con signo
5      idivq   %rcx                        # y/z
6      jmp     .L6                        # jmp a merge
7  .L9:                                # case 3
8      movl    $1, %eax                    # w = 1
9  .L6:                                # merge:
10     addq    %rcx, %rax                   # w += z
11     ret

```

Code blocks: $x == 5, 6$, default

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

```
1  .L7:                                # case 5, 6  
2      movl  $1, %eax                 # w = 1  
3      subq  %rdx, %rax               # w -= z  
4      ret  
5  .L8:                                # default  
6      movl  $2, %eax                 # w = 2  
7      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`

