

Programación a Nivel-Máquina II: Control

Estructura de Computadores
Semana 4

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)

- Control
 - § 3.6 pp.236-274

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

- Probl. 3.13 – 3.14 § 3.6.2, pp.240, 241
- **Probl. 3.15** § 3.6.4, pp.245[†]
- Probl. 3.16 – 3.18 § 3.6.5, pp.248₂, 249
- Probl. **3.19** – 3.21 § 3.6.6, pp.**252[‡]**, 255₂
- Probl. 3.22 – 3.29 § 3.6.7, pp.257, 258, 260, 262, 264, 267₂, 268
- Probl. 3.30 – 3.31 § 3.6.8, pp.272, 273

Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/**C.1 BRY com**

[†] direccionamiento relativo a contador de programa, "PC-relative"

[‡] penalización por predicción saltos 2

Programación Máquina II: Control

- **Control: Códigos de condición**
- Saltos condicionales
- Bucles
- Sentencias switch

Estado del Procesador (x86-64, Parcial)

■ Información sobre el programa ejecutándose actualmente

- Datos temporales (**%rax**, ...)
- Situación de la pila en tiempo de ejecución[†] (**%rsp**)
- Situación actual del contador de programa (**%rip**)
- Estado de comparaciones recientes (**CF, ZF, SF, OF**)

Registros de propósito general

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

Tope de pila actual

%rip **Puntero de instrucción[‡]**

CF ZF SF OF **Códigos de condición**

[‡] "instruction pointer", de ahí %rip

[†] "runtime stack"

Códigos de Condición (su ajuste implícito)

■ Registros de un solo bit[†]

- **CF** Flag de Acarreo (p/sin signo) **SF** Flag de Signo (para ops. con signo)
- **ZF** Flag de Cero **OF** Flag de Overflow[‡] (ops. con signo)

■ Ajustados implícitamente por las operaciones aritméticas (interpretarlo como efecto colateral)

Ejemplo: `addq Src, Dest` \leftrightarrow `t = a+b`

CF puesto a 1 sii sale acarreo del bit más significativo (desbord. op. sin signo)

ZF a 1 sii `t == 0`

SF a 1 sii `t < 0` (como número con signo)

OF a 1 sii desbord. en complemento a dos (desbord. op. con signo)
(`a > 0 && b > 0 && t < 0`) || (`a < 0 && b < 0 && t >= 0`)

■ No afectados por la instrucción `lea`

[†] “flag” = “bandera”, deberíamos traducir “flag” por “indicador”, pero se suele dejar así.
“optimization flags” debería ser “modificador/conmutador”, también se suele dejar así.

[‡] “overflow” = “desbordamiento”, y los otros
“carry/zero/sign” = “acarreo/cero/signo” 5

Códigos de Condición (ajuste explícito: Compare)

■ Ajuste Explícito mediante la Instrucción Compare

- `cmpq Src2, Src1`

- `cmpq b,a` equivale a restar `a-b` pero sin ajustar el destino

- **CF a 1** sii sale acarreo del MSB[†] (c_n) (hacer caso cuando `cmp` sin signo)

- **ZF a 1** sii `a == b`

- **SF a 1** sii `(a-b) < 0` (como número con signo)

- **OF a 1** sii overflow[‡] en complemento a dos (atender si `cmp` con signo)
`(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`
definición overflow $OF = (c_n \wedge c_{n-1})$ mientras que acarreo $CF = c_n$

[†] “MSB” = bit más significativo

[‡] dejar sin traducir “overflow” ayuda didácticamente a distinguirlo del acarreo (desbordamiento sin signo) al recordar los flags OF/CF 6

Códigos de Condición (ajuste explícito: Test)

■ Ajuste Explícito mediante la Instrucción Test

- `testq Src2, Src1`
- `testq b,a` equivale a hacer `a&b` pero sin ajustar el destino

■ **ZF a 1** sii $(a \& b) == 0$

■ **SF a 1** sii $(a \& b) < 0$

- Ajusta los códigos de condición según el valor de *Src1* & *Src2*
- Útil cuando uno de los operandos es una máscara
- Para comprobar si un valor es 0, gcc usa `testq`, no `cmpq`
`cmpq $0, %rax`
`testq %rax, %rax`

Consultando Códigos de Condición

■ Instrucciones SetCC *Dest*

- Ajustar el byte destino a 0/1 según el código de condición indicado con CC[†] (combinación de flags deseada)
- ***Dst registro*** debe ser tamaño byte, ***Dst memoria*** sólo se modifica 1^{er} LSByte[†]

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)

“CC” = “condition code”

† “LSByte” = byte menos significativo,⁸

Registros enteros 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

- Se puede referenciar el LSByte, tiene nombre de registro propio

Consultando Códigos de Condición (Cont.)

■ Instrucciones SetCC *Dest*

- Ajustar un byte suelto *Dest* según el código de condición

■ Uno de los registros byte direccionables

- No se alteran los restantes bytes
- Típicamente se usa `movzbl`[†] para terminar trabajo
 - las instrucciones de 32-bit también ponen los 32 MSB a 0

```
int gt (long x, long y)
{
    return x > y;
}
```

Registro	Uso(s)
%rdi	Argumento <i>x</i>
%rsi	Argumento <i>y</i>
%eax	Valor de retorno

```
gt:
    cmpq    %rsi, %rdi    # Comparar x:y
    setg    %al           # Poner a 1 si >
† movzbl   %al, %eax     # Resto %rax a cero
    ret
```

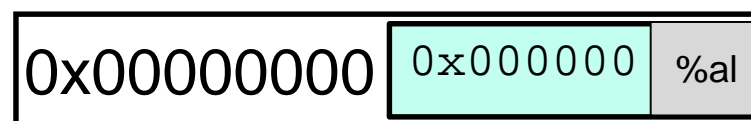
[†] "Move with Zero-extend Byte to Long"
mnemotécnico MOVZX según Intel 10

Consultando Códigos de Condición (Cont.)

■ Instrucciones SetCC *Dest*

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

`movzbl %al, %eax`



gt:

Puesto todo a 0

† `movzbl %al, %eax # Resto %rax a cero`
`ret`

† “Move with Zero-extend Byte to Long”
mnemotécnico MOVZX según Intel 11

Programación Máquina II: Control

- Control: Códigos de condición
- Saltos condicionales
- Bucles
- Sentencias switch

Salto

■ Instrucciones jCC

- Saltar a otro lugar del código si se cumple el código de condición CC

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

Ejemplo de Salto Condicional (al viejo estilo)

■ Generación[†]

```
shark> gcc -Og -S -fno-if-conversion control.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    # 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 el gcc-7.3.0 que viene con Ubuntu-18.04

-fif-conversion activada para -O123s pero no para -Og 14

Expresándolo con código Goto

- C permite la sentencia goto
- 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 (usando saltos)

Código C

```
val = Test ? Then_Expr : Else_Expr;
```

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

Versión Goto

```
ntest = !Test;  
if (ntest) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

- Crear regiones de código separadas para las expresiones Then y Else
- Ejecutar sólo la adecuada

Usando Movimientos Condicionales

■ Instrucciones Mvmt. Condicional

- Las instrucciones implementan:
 $\text{if (Test) Dest} \leftarrow \text{Src}$
- En procesadores x86 posteriores a 1995
(Pentium Pro/II)
- GCC intenta utilizarlas
 - Pero sólo cuando sepa que es seguro

■ ¿Por qué?

- Ramificaciones muy perjudiciales para flujo instrucciones en cauces[†]
- Movimiento condicional no requiere transferencia de control

Código C

```
val = Test  
    ? Then_Expr  
    : Else_Expr;
```

Versión Goto

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

Ejemplo de Movimiento Condicional[†]

```
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 control.c
ó incluso con gcc -O -S control.c
en el gcc-7.3.0 que viene con Ubuntu-18.04 18

Malos Casos para Movimientos Condicionales

- Recordar que se calculan ambos valores

Cálculos costosos

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

- Sólo tiene sentido cuando son cálculos muy sencillos

Cálculos arriesgados

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

- Pueden tener efectos no deseables

Cálculos con efectos colaterales

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

- No deberían tener efectos colaterales

Ejercicio

cmpq b,a equiv. restar t=a-b pero sin ajustar destino

- **CF a 1** sii $c_n == 1$, porque $a < b$ sin signo
- **ZF a 1** sii $t == 0$, porque $a == b$
- **SF a 1** sii $t_n == 1$, porque $a < b$ con signo
- **OF a 1** sii $(c_n \wedge c_{n-1}) == 1$, pq. $a-b$ (signo) mal hecha

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

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

%rax	SF	CF	OF	ZF

Notar: **setl** y **movzbl** no ajustan códigos de condición

Ejercicio

cmpq b,a equiv. restar t=a-b pero sin ajustar destino

- **CF a 1** sii $c_n == 1$, porque $a < b$ sin signo
- **ZF a 1** sii $t == 0$, porque $a == b$
- **SF a 1** sii $t_n == 1$, porque $a < b$ con signo
- **OF a 1** sii $(c_n \wedge c_{n-1}) == 1$, pq. $a-b$ (signo) mal hecha

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

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

%rax	SF	CF	OF	ZF
0x0000 0000 0000 0000	0	0	0	1
0xFFFF FFFF FFFF FFFF	1	1	0	0
0xFFFF FFFF FFFF FFFF	1	0	0	0
0xFFFF FFFF FFFF FF01	1	0	0	0
0x0000 0000 0000 0001	1	0	0	0

Notar: **setl** y **movzbl** no ajustan códigos de condición

Programación Máquina II: Control

- Control: Códigos de condición
- Saltos condicionales
- **Bucles**
- Sentencias switch

Ejemplo de bucle “Do-While”

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;
}
```

- Contar el número de 1's en el argumento x (“popcount”[†])
- Usar salto condicional para seguir iterando o salir del bucle

[†] “population count”= peso Hamming, distancia Hamming (al 0), suma lateral... 23

Compilación del bucle “Do-While”

Versión Goto

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

Registro	Uso(s)
%rdi	Argumento x
%rax	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 Athlon 64 (2003-2005) en RET 1B tras flowctrl.
Software Optimization Guide for AMD64 Family 10-12h (2010-2011) recomienda RET 0*

Guide for AMD64 Family 15h (2011) no menciona ningún problema

Traducción en General de “Do-While”

Código C

```
do  
    Body  
while ( Test );
```



Versión Goto

```
loop:  
    Body  
    if ( Test )  
        goto loop
```

■ **Body:** {
 *Sentencia*₁;
 *Sentencia*₂;
 ...
 *Sentencia*_n;
}

“body” = cuerpo,
“statement” = sentencia,
“test” = comprobación. 25

Traducción en General de “While” (#1)

Código C

```
while ( Test )  
    Body
```



Versión Goto

```
    goto test;  
loop:  
    Body  
test:  
    if ( Test )  
        goto loop;  
done:
```

- Traducción tipo “salta-en-medio” †
- Usada con -O0 / -Og

Ejemplo de bucle “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;
}
```

- Comparar con la versión do-while de la misma función
- El goto inicial empieza el bucle por **test** (“en medio”)

[†] *pcount_goto_jtm* viene de “jump-to-middle” 27

Traducción en General de “While” (#2)

Versión While

```
while ( Test )  
    Body
```



Versión Do-While

```
if ( ! Test )  
    goto done;  
do  
    Body  
    while( Test );  
done:
```



- Traducción tipo “**copia-test**”
 - Conversión a “do-while”
- Usada con **-O1[†]**

Versión Goto

```
if ( ! Test )  
    goto done;  
loop:  
    Body  
    if ( Test )  
        goto loop;  
done:
```

[†] evitar test al principio con gcc -O123 -fno-tree-ch

ó con gcc -Os -fno-reorder-blocks 28

Ejemplo de bucle “While” (#2)

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;
}
```

- Comparar con la versión do-while de la misma función
- El primer condicional guarda la entrada al bucle

Forma del bucle “For”

Forma General

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

```
#define WSIZE 8*sizeof(long)  
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;  
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

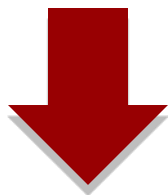
```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

“Init” = inicialización,
“test” = comprobación,
“update” = actualización,
“body” = cuerpo. 30

Bucle “For” → Bucle While

Versión For

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



Versión While

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

Conversión For-While

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

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

```
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 Bucle “For” 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;
}
```

Versión Goto

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

Init

! Test

Body

Update

Test

- La comprobación inicial se puede optimizar (quitándola)

Programación Máquina II: Control

- Control: Códigos de condición
- Saltos condicionales
- Bucles
- Sentencias switch[†]

[†] “switch”=conmutador,
tampoco traducimos
“for” o “while” 34

```
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;
}
```

Ejemplo de sentencia switch

- Múltiples etiquetas de caso
 - Aquí: 5 y 6
- Caídas en cascada[†]
 - Aquí: 2
- Casos ausentes[†]
 - Aquí: 4

[†] “fall-through cases”, “missing cases” 35

Estructura de una Tabla de Saltos

Forma switch

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

Tabla Saltos[†]

JTab:

Targ0
Targ1
Targ2
•
•
•
Targn-1

Destinos salto[†]

Targ0:

**Code Block
0**

Targ1:

**Code Block
1**

Targ2:

**Code Block
2**

•
•
•

Targn-1:

**Code Block
n-1**

Traducción aprox. (C ficticio, gcc supp. jump tables)

```
goto *JTab[x];
```

Ejemplo de 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?

Ejemplo de Sentencia Switch

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

Tabla de saltos[†]

```
.section      .rodata
    .align 8
.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
```

Inicialización:

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

 ***Salto indirecto***

Explicación Inicialización Ensamblador

■ Estructura de la Tabla

- Cada destino salto requiere 8 bytes
- Dirección base es .L4

■ Saltos

- **Directo:** `jmp .L8`
- Destino salto indicado por etiqueta .L8
- **Indirecto:** `jmp *.L4(,%rdi,8)`
- Inicio de la tabla de saltos: .L4
- Se debe escalar por un factor de 8 (direcciones ocupan 8 bytes)
- Captar destino salto desde la Dirección Efectiva $.L4 + x \cdot 8$
 - Sólo para $0 \leq x \leq 6$

Tabla de saltos

```
.section      .rodata
    .align 8
.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
```

Tabla de Saltos

Tabla de saltos

```
.section .rodata
.align 8
.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:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```


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

Tratamiento de Caídas en Cascada

```
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;

Bloques de Código (x == 2, x == 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
    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

mnemotécnico CQO según Intel

† “Convert Quad to Oct”

Bloques de Código (x == 5, x == 6, default)

```
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)

Guía de trabajo autónomo (4h/s)

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

- Control
 - § 3.6 pp.236-274

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

- Probl. 3.13 – 3.14 § 3.6.2, pp.240, 241
- Probl. **3.15** § 3.6.4, **pp.245**[†]
- Probl. 3.16 – 3.18 § 3.6.5, pp.248₂, 249
- Probl. **3.19** – 3.21 § 3.6.6, **pp.252**[‡], 255₂
- Probl. 3.22 – 3.29 § 3.6.7, pp.257, 258, 260, 262, 264, 267₂, 268
- Probl. 3.30 – 3.31 § 3.6.8, pp.272, 273

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/**C.1 BRY com**

[†] direccionamiento relativo a contador de programa, "PC-relative"

[‡] penalización por predicción saltos 46