

# Programación a Nivel-Máquina II: Aritmética & Control

Estructura de Computadores  
Semana 3

## Bibliografía:

[BRY11] Cap.3      Computer Systems: A Programmer's Perspective. Bryant, O'Hallaron. Pearson, 2011  
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

# Guía de trabajo autónomo (4h/s)

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

- Arithmetic and Logical Operations, Control. (hasta 3.6.7, Switch Statements)
  - 3.5 - 3.6 pp.211-247 (sec.3.6.7 en siguiente lección)

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

- Probl. 3.6 - 3.27 pp.212-16, 218, 222-23, 226, 229-30, 232-33, 235-36, 239-40, 243, 246

## Bibliografía:

[BRY11] Cap.3

Computer Systems: A Programmer's Perspective. Bryant, O'Hallaron. Pearson, 2011

Signatura ESIT/C.1 BRY com

# Programación Máquina II: Aritmética/Control

- **Modo direccionamiento completo, cálculo de direcciones (leal)**
- Operaciones aritméticas
- Control: Códigos de condición
- Saltos condicionales
- Bucles while

# Modos Direcccionamiento a memoria completos

## ■ Forma más general

$$\text{D(Rb,Ri,S)}$$

$$\text{Mem[Reg[Rb]+S*Reg[Ri]+ D]}$$

- D: “Desplazamiento” constante 1, 2, ó 4 bytes
- Rb: Registro base: Cualquiera de los 8 registros enteros
- Ri: Registro índice: Cualquiera, excepto %esp
  - Tampoco es probable que se use %ebp
- S: Factor de escala: 1, 2, 4, ú 8 (¿por qué esos números?)

## ■ Casos Especiales

$$\text{(Rb,Ri)}$$

$$\text{Mem[Reg[Rb]+Reg[Ri]]}$$

$$\text{D(Rb,Ri)}$$

$$\text{Mem[Reg[Rb]+Reg[Ri]+D]}$$

$$\text{(Rb,Ri,S)}$$

# Ejemplos de Cálculo de Direcciones

%edx	0xf000
%ecx	0x0100

Expresión	Cálculo de Dirección	Dirección
0x8(%edx)		
(%edx,%ecx)		
(%edx,%ecx,4)		
0x80(,%edx,2)		

# Instrucción para el Cálculo de Direcciones

## ■ **leal Src, Dest \***

- Src es cualquier expresión de modo direccionamiento (a memoria)
- Ajusta Dest a la dirección indicada por la expresión

## ■ **Usos**

- Calcular direcciones sin hacer referencias a memoria
  - p.ej., traducción de  $p = \&x[i]$ ;
- Calcular expresiones aritméticas de la forma  $x + k*y$ 
  - $k = 1, 2, 4, \text{ ó } 8$

## ■ **Ejemplo**

```
int mul12(int x)
{
    return x*12;
}
```

Traducción a ASM por el compilador:

```
leal (%eax,%eax,2), %eax ;t <- x+x*2
sall $2, %eax           ;return t<<2
```

\* "source/destination" = fuente/destino 6

# Programación Máquina II: Aritmética/Control

- Modo direccionamiento completo, cálculo de direcciones (leal)
- **Operaciones aritméticas**
- Control: Códigos de condición
- Saltos condicionales
- Bucles while

# Algunas operaciones aritméticas

## ■ Instrucciones de dos operandos:

### Formato

### Operación \*

addl	Src, Dest	Dest = Dest + Src
subl	Src, Dest	Dest = Dest – Src
imull	Src, Dest	Dest = Dest * Src
sall	Src, Dest	Dest = Dest << Src
sarl	Src, Dest	Dest = Dest >> <sub>A</sub> Src
shrl	Src, Dest	Dest = Dest >> Src
xorl	Src, Dest	Dest = Dest ^ Src
andl	Src, Dest	Dest = Dest & Src
orl	Src, Dest	Dest = Dest   Src

También llamada shll  
Aritméticas

← Lógicas →

## ■ ¡Cuidado con el orden de los argumentos!

## ■ No se distingue entre enteros con/sin signo (*¿por qué?*)

\* “source/destination” = fuente/destino 8



# Algunas operaciones aritméticas

## ■ Instrucciones de un operando:

Formato		Operación
<code>incl</code>	<code>Dest</code>	$\text{Dest} = \text{Dest} + 1$
<code>decl</code>	<code>Dest</code>	$\text{Dest} = \text{Dest} - 1$
<code>negl</code>	<code>Dest</code>	$\text{Dest} = -\text{Dest}$
<code>notl</code>	<code>Dest</code>	$\text{Dest} = \sim\text{Dest}$

## ■ Para más instrucciones consultar el libro

# Ejemplo de expresiones aritméticas

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

arith:

```
pushl    %ebp
movl     %esp, %ebp
```

} Ajuste  
Inicial

```
movl     8(%ebp), %ecx
movl     12(%ebp), %edx
leal     (%edx,%edx,2), %eax
sall     $4, %eax
leal     4(%ecx,%eax), %eax
addl     %ecx, %edx
addl     16(%ebp), %edx
imull    %edx, %eax
```

} Cuerpo

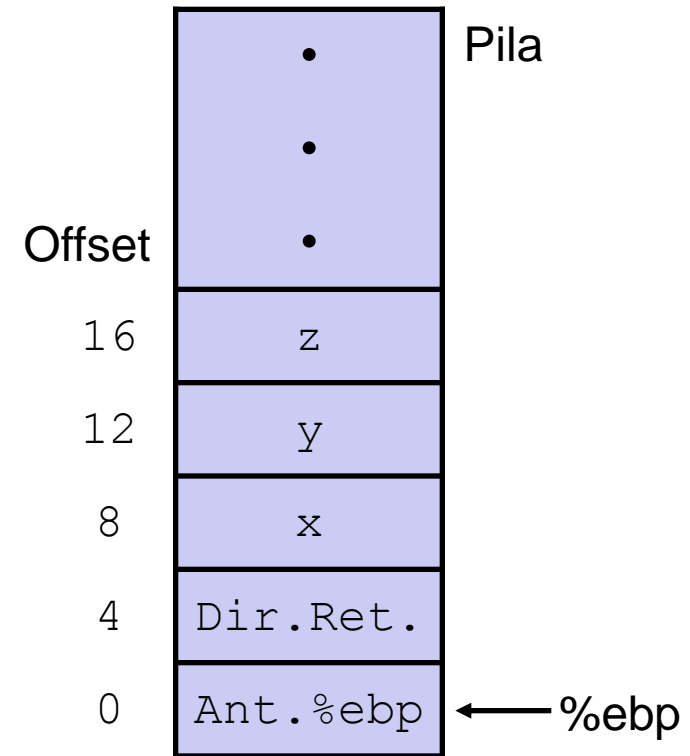
```
popl     %ebp
ret
```

} Fin

# Comprendiendo arith

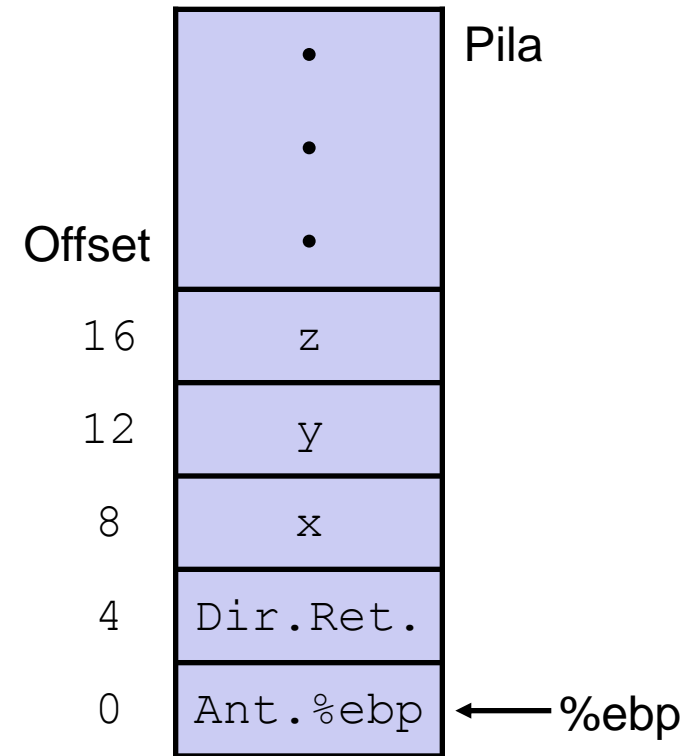
```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
movl    8(%ebp), %ecx
movl    12(%ebp), %edx
leal    (%edx,%edx,2), %eax
sall    $4, %eax
leal    4(%ecx,%eax), %eax
addl    %ecx, %edx
addl    16(%ebp), %edx
imull   %edx, %eax
```



# Comprendiendo arith

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```



movl	8(%ebp), %ecx	# ecx = x
movl	12(%ebp), %edx	# edx = y
leal	(%edx,%edx,2), %eax	# eax = y*3
sall	\$4, %eax	# eax *= 16 (t4)
leal	4(%ecx,%eax), %eax	# eax = t4 +x+4 (t5)
addl	%ecx, %edx	# edx = x+y (t1)
addl	16(%ebp), %edx	# edx += z (t2)
imull	%edx, %eax	# eax = t2 * t5 (rval)

# Observaciones sobre arith

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instrucciones en orden distinto que en el código C
- Algunas exprs. requieren varias instrucciones
- Algunas instrucciones cubren varias expresiones
- Se obtiene exactamente el mismo código al compilar:
- $(x+y+z) * (x+4+48*y)$

movl	8(%ebp), %ecx	# ecx = x
movl	12(%ebp), %edx	# edx = y
leal	(%edx,%edx,2), %eax	# eax = y*3
sall	\$4, %eax	# eax *= 16 (t4)
leal	4(%ecx,%eax), %eax	# eax = t4 +x+4 (t5)
addl	%ecx, %edx	# edx = x+y (t1)
addl	16(%ebp), %edx	# edx += z (t2)
imull	%edx, %eax	# eax = t2 * t5 (rval)

# Otro Ejemplo

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

logical:

pushl %ebp	}	Ajuste Inicial
movl %esp,%ebp		
movl 12(%ebp),%eax	}	Cuerpo
xorl 8(%ebp),%eax		
sarl \$17,%eax		
andl \$8185,%eax		
popl %ebp	}	Fin
ret		

movl 12(%ebp),%eax	# eax = y
xorl 8(%ebp),%eax	# eax = x^y (t1)
sarl \$17,%eax	# eax = t1>>17 (t2)
andl \$8185,%eax	# eax = t2 & mask (rval)

# Otro Ejemplo

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

logical:

```
pushl %ebp
movl %esp,%ebp
```

} Ajuste  
Inicial

```
movl 12(%ebp),%eax
xorl 8(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

} Cuerpo

```
popl %ebp
ret
```

} Fin

```
movl 12(%ebp),%eax
xorl 8(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

```
# eax = y
# eax = x^y      (t1)
# eax = t1>>17   (t2)
# eax = t2 & mask (rval)
```

# Otro Ejemplo

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

logical:

pushl %ebp	}	Ajuste Inicial
movl %esp,%ebp		
movl 12(%ebp),%eax	}	Cuerpo
xorl 8(%ebp),%eax		
sarl \$17,%eax		
andl \$8185,%eax		
popl %ebp	}	Fin
ret		

movl 12(%ebp),%eax	# eax = y
xorl 8(%ebp),%eax	# eax = x^y (t1)
sarl \$17,%eax	# eax = t1>>17 (t2)
andl \$8185,%eax	# eax = t2 & mask (rval)



# Otro Ejemplo

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

$$2^{13} = 8192, 2^{13} - 7 = 8185$$

logical:

```
pushl %ebp
movl %esp,%ebp
```

} Ajuste  
Inicial

```
movl 12(%ebp),%eax
xorl 8(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

} Cuerpo

```
popl %ebp
ret
```

} Fin

```
movl 12(%ebp),%eax
xorl 8(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

```
# eax = y
# eax = x^y      (t1)
# eax = t1>>17   (t2)
# eax = t2 & mask (rval)
```

# Programación Máquina II:

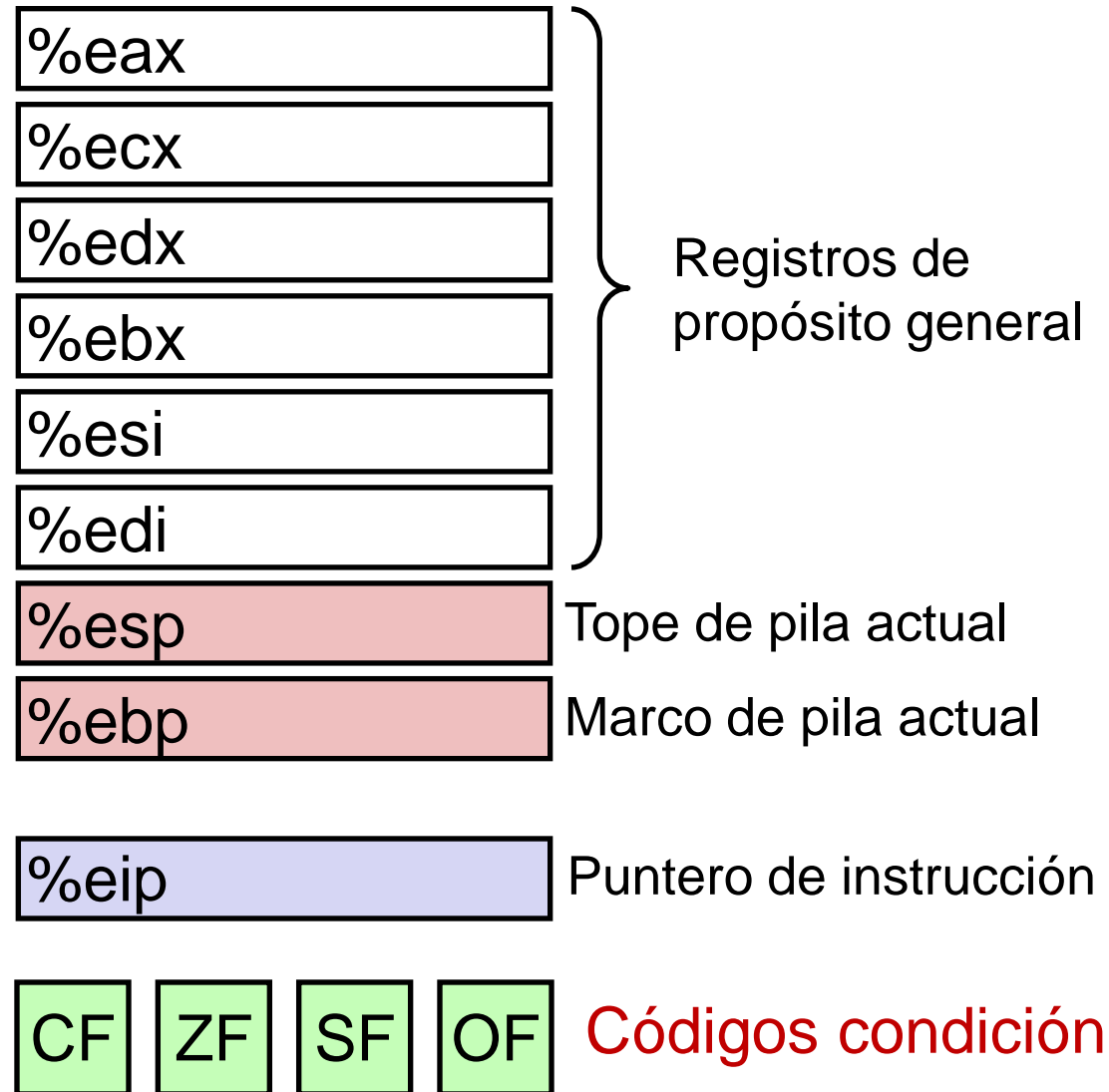
## Aritmética/Control

- Modo direccionamiento completo, cálculo de direcciones (leal)
- Operaciones aritméticas
- **Control: Códigos de condición**
- Saltos condicionales
- Bucles while

# Estado del procesador (IA32, Parcial)

## ■ Información sobre el programa ejecutándose actualmente

- Datos temporales ( %eax, ... )
- Situación de la pila en tiempo de ejecución\* ( %ebp,%esp )
- Situación actual del contador de programa ( %eip, ... )
- Estado de comparaciones recientes ( CF, ZF, SF, OF )



# Códigos de condición (su ajuste implícito)

## ■ Registros de un solo bit\*

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

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

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

CF puesto a 1    sii 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 2 (desbord.\*\* op. con signo)  
(`a > 0 && b > 0 && t < 0`) || (`a < 0 && b < 0 && t >= 0`)

## ■ No afectados por la instrucción `lea`

## ■ Documentación completa (IA32), enlace en la web de la asignatura

\* "flag" = "bandera", "indicador"

\*\* "overflow" = "desbordamiento",  
"carry/zero/sign" = "acarreo/cero/signo" 20

# Códigos de condición (ajuste explícito: Compare)

## ■ Ajuste explícito mediante la Instrucción Compare

- `cmpl/cmpq Src2, Src1`
- `cmpl 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 comp. 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 (consultar si comp. con signo)  
`(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`  
definición alternativa overflow ( $c_n \wedge c_{n-1}$ )

\* “msb” = bit más significativo

\*\* dejar sin traducir “overflow” ayuda didácticamente a distinguirlo del acarreo (desbord. sin signo) al recordar los flags OF/CF 21

# Códigos de condición (ajuste explícito: Test)

## ■ Ajuste explícito mediante la Instrucción Test

- `testl/testq Src2, Src1`
- `testl b,a` equivale a hacer `a&b` pero sin ajustar el destino
- Ajusta los códigos de condición según el valor de Src1 & Src2
- Útil cuando uno de los operandos es una máscara
- **ZF a 1** sii `a&b == 0`
- **SF a 1** sii `a&b < 0`

# Consultando códigos de condición

## ■ Instrucciones SetX

- Ajustar un byte suelto según el código de condición (combinación deseada)

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

# Consultando códigos de condición

## ■ Instrucciones SetX :

- Ajustar un byte suelto según el código condición

## ■ Uno de los 8 registros byte direccionables

- No se alteran los restantes 3 bytes
- Típicamente se usa `movzbl*` para terminar trabajo

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

## Cuerpo

```
movl 12(%ebp), %eax    # eax = y
cmpl %eax, 8(%ebp)     # Compare x : y
setg %al               # al = x > y
movzbl %al, %eax       # Zero rest of %eax
```

%eax	%ah	%al
------	-----	-----

%ecx	%ch	%cl
------	-----	-----

%edx	%dh	%dl
------	-----	-----

%ebx	%bh	%bl
------	-----	-----

%esi
------

%edi
------

%esp
------

%ebp
------

\* "Move with Zero-extend Byte to Long", mnemotécnico MOVZX según Intel



# Consultando códigos de condición: x86-64

## ■ Instrucciones SetX :

- Ajustar un byte suelto según el código de condición
- No se alteran los restantes 3 bytes

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

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

## Cuerpos

```
cmpl %esi, %edi
setg %al
movzbl %al, %eax
```

```
cmpq %rsi, %rdi
setg %al
movzbl %al, %eax
```

¿Es %rax cero?

¡Sí: Las instrucciones de 32-bit ponen los (otros) 32 bits más significativos a 0!

# Programación Máquina II:

## Aritmética/Control

- Modo direccionamiento completo, cálculo de direcciones (leal)
- Operaciones aritméticas
- Control: Códigos de condición / x86-64
- **Salto (y Movimientos) condicionales**
- Bucles

# Saltos

## ■ Instrucciones jX

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

jX	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^OF)& $\sim$ ZF	Greater (signo)
jge	$\sim$ (SF^OF)	Greater or Equal (signo)
jl	(SF^OF)	Less (signo)
jle	(SF^OF) ZF	Less or Equal (signo)
ja	$\sim$ CF& $\sim$ ZF	Above (sin signo)
jb	CF	Below (sin signo)

# Ejemplo de salto condicional

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

```
absdiff:
    pushl    %ebp
    movl     %esp, %ebp
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    cmpl     %eax, %edx
    jle      .L6
    subl     %eax, %edx
    movl     %edx, %eax
    jmp      .L7
.L6:
    subl     %edx, %eax
.L7:
    popl     %ebp
    ret
```

Ajuste Inicial

Cuerpo1

Cuerpo2a

Cuerpo2b

Fin

# Ejemplo de salto condicional

```
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    goto Exit;
Else:
    result = y-x;
Exit:
    return result;
}
```

## ■ C permite “goto” como medio transfer. control

- Más próximo al estilo de programación a nivel-máquina

## ■ Generalmente considerado mal estilo de programación

absdiff:

```
    pushl    %ebp
    movl     %esp, %ebp
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    cmpl     %eax, %edx
    jle      .L6
    subl     %eax, %edx
    movl     %edx, %eax
    jmp      .L7
```

.L6:

```
    subl     %edx, %eax
```

.L7:

```
    popl     %ebp
    ret
```

} Ajuste Inicial

} Cuerpo1

} Cuerpo2a

} Cuerpo2b

} Fin

# Ejemplo de salto condicional

```
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    goto Exit;
Else:
    result = y-x;
Exit:
    return result;
}
```

absdiff:

```
    pushl    %ebp
    movl     %esp, %ebp
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    cmpl     %eax, %edx
    jle      .L6
    subl     %eax, %edx
    movl     %edx, %eax
    jmp      .L7
.L6:
    subl     %edx, %eax
.L7:
    popl     %ebp
    ret
```

Ajuste Inicial

Cuerpo1

Cuerpo2a

Cuerpo2b

Fin

# Ejemplo de salto condicional

```
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    goto Exit;
Else:
    result = y-x;
Exit:
    return result;
}
```

absdiff:

pushl	%ebp	}	Ajuste Inicial
movl	%esp, %ebp		
movl	8(%ebp), %edx	}	Cuerpo1
movl	12(%ebp), %eax		
cmpl	%eax, %edx		
jle	.L6	}	Cuerpo2a
subl	%eax, %edx		
movl	%edx, %eax		
jmp	.L7		
.L6:			
subl	%edx, %eax	}	Cuerpo2b
.L7:			
popl	%ebp	}	Fin
ret			

# Ejemplo de salto condicional

```
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    goto Exit;
Else:
    result = y-x;
Exit:
    return result;
}
```

absdiff:

```
    pushl    %ebp
    movl     %esp, %ebp
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    cmpl     %eax, %edx
    jle      .L6
    subl     %eax, %edx
    movl     %edx, %eax
    jmp      .L7
.L6:
    subl     %edx, %eax
.L7:
    popl     %ebp
    ret
```

Ajuste Inicial

Cuerpo1

Cuerpo2a

Cuerpo2b

Fin



# Traducción en general expresión condicional

## Código C

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

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

## Versión Goto

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

- Test\* es expresión devolviendo entero
  - = 0 interpretado como falso
  - ≠ 0 interpretado como verdadero
- Crear regiones código (separadas) para las expresiones Then & Else
- Ejecutar sólo la adecuada

\* “test” = “prueba”, aquí significa “comprobación/condición” 33

# Usando movimientos condicionales

## ■ Instrucciones mov. condicional

- Son instrucciones que implementan:  
 $\text{if (Test) Dest} \leftarrow \text{Src}$
- En procesadores x86 posteriores a 1995  
(Pentium Pro/II)
- GCC no siempre las usa
  - Intenta preservar compatibilidad con procesadores antiquísimos
  - Habilitadas para x86-64
  - Usar `switch*` `-march=686` para IA32

## ■ ¿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

```
tval = Then_Expr;  
result = Else_Expr;  
t = Test;  
if (t) result = tval;  
return result;
```

\*\* “pipeline” = “tubería”, aquí en sentido “segmentación de cauce”

\* “switch” = “conmutador”, aquí sería “modificador” (en plural, “switches” de compilación) 34

# Ej. movimiento condicional: x86-64

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

absdiff:

x en	movl	%edi, %edx	
%edi	subl	%esi, %edx	# tval = x-y
	movl	%esi, %eax	
y en	subl	%edi, %eax	# result = y-x
%esi	cmpl	%esi, %edi	# Compare x:y
	<b>cmovg</b>	<b>%edx, %eax</b>	<b># If &gt;, result = tval</b>
	ret		

# Malos casos para mov. condicionales

## Cálculos costosos

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

- Se calculan ambos valores
- Sólo tiene sentido cuando son cálculos muy sencillos

## Cálculos arriesgados

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

- Puede tener efectos no deseables

## Cálculos con efectos colaterales

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

- No debería tener efectos colaterales

# Programación Máquina II: Aritmética/Control

- Modo direccionamiento completo, cálculo de direcciones (leal)
- Operaciones aritméticas
- Control: Códigos de condición / x86-64
- Saltos (y Movimientos) condicionales
- **Bucles**

# Ejemplo de bucle “Do-While”

## Código C

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

## Versión Goto

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

- Contar el número de 1's en el argum. x (“popcount” \*)
- Usar salto condicional para, o bien continuar dando vueltas, o salir del bucle

# Compilación del bucle “Do-While”

## Versión Goto

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

### ■ Registros:

%edx	x
%ecx	result

```
movl    $0, %ecx        # result = 0
.L2:
movl    %edx, %eax
andl    $1, %eax        # t = x & 1
addl    %eax, %ecx      # result += t
shrl    %edx            # x >>= 1
jne     .L2             # If !0, goto loop
```

# Traducción en general de “Do-While”

## Código C

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

## Versión Goto

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

■ **Body**\*: {  
    Sentencia<sub>1</sub>;  
    Sentencia<sub>2</sub>;  
    ...  
    Sentencia<sub>n</sub>;  
}

## ■ Test devuelve entero

- = 0 interpretado como falso
- ≠ 0 interpretado como verdadero



# Ejemplo de bucle “While”

## Código C

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

## Versión Goto

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

- ¿Es este código equivalente a la versión do-while?

# Traducción en general de “While”

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:
```

# Ejemplo de bucle “For”

## Código C

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

- ¿Es este código equivalente a las otras versiones?

# Forma de los bucles “For”

## Forma General

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

```
for (i = 0; i < WSIZE; i++) {  
    unsigned mask = 1 << i;  
    result += (x & mask) != 0;  
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned mask = 1 << i;  
    result += (x & mask) != 0;  
}
```

# Bucle “For” → Bucle While

Versión For

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



Versión While

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

# Bucle “For” → ... → Goto

Versión For

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

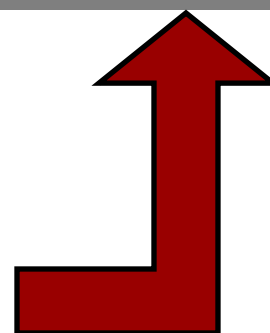


Versión While

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



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



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

# Ejemplo conversión Bucle “For”

## Código C

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

- La comprobación inicial se puede optimizar—quitándola

## Versión Goto

```
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    i = 0;
    if (!(i < WSIZE))
    goto done;
loop:
    {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    i++;
    if (i < WSIZE)
        goto loop;
done:
    return result;
}
```

*Init*

*! Test*

*Body*

*Update*

*Test*

# Programación Máquina II: Resumen

## ■ Aritmética & Control

- Modo direccionamiento completo, cálculo de direcciones (leal)
- Operaciones aritméticas
- Control: Códigos de condición
- Saltos condicionales y movimientos condicionales
- Bucles

## ■ Siguiente lección (Switch & Procedimientos)

- Sentencias switch
- Pila
- Llamada / retorno
- Disciplina de llamada a procedimientos



# Guía de trabajo autónomo (4h/s)

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

- Arithmetic and Logical Operations, Control. (hasta 3.6.7, Switch Statements)
  - 3.5 - 3.6 pp.211-247 (sec.3.6.7 en siguiente lección)
  - Probl. 3.6 - 3.27 pp.212-16, 218, 222-23, 226, 229-30, 232-33, 235-36, 239-40, 243, 246

## Bibliografía:

[BRY11] Cap.3

Computer Systems: A Programmer's Perspective. Bryant, O'Hallaron. Pearson, 2011

Signatura ESIT/C.1 BRY com