# O'Hallaron: CS:APP, 2ª Ed.

## Signatura: ESIIT/C.1 BRY com

### Capítulo 3: Representación de Programas a nivel de máquina

***Problemas Prácticos T2.2:***

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

*3.48-3.49, pp.312,315*

**3.6.** Suponer que el registro %eax contiene el valor x y %ecx contiene el valor y. Rellenar la tabla de abajo con fórmulas indicando el valor que se almacenará en el registro %edx para cada una de las instrucciones de código ensamblador dadas.

| Instrucción | Resultado |
|---|---|
| `leal 6(%eax), %edx` | |
| `leal (%eax,%ecx), %edx` | |
| `leal (%eax,%ecx,4), %edx` | |
| `leal 7(%eax,%eax,8), %edx` | |
| `leal 0xA(,%ecx,4), %edx` | |
| `leal 9(%eax,%ecx,2), %edx` | |

**3.7.** Asumir que los siguientes valores están almacenados en los registros y direcciones de memoria que se indican:

| Dirección | Valor |
|---|---|
| 0x100 | 0xFF |
| 0x104 | 0xAB |
| 0x108 | 0x13 |
| 0x10C | 0x11 |

| Registro | Valor |
|---|---|
| %eax | 0x100 |
| %ecx | 0x1 |
| %edx | 0x3 |

Rellenar la siguiente tabla mostrando los efectos de las siguientes instrucciones, tanto en términos del registro o posición de memoria que será actualizado como del valor resultante:

| Instrucción | Destino | Valor |
|---|---|---|
| `addl %ecx,(%eax)` | | |
| `subl %edx,4(%eax)` | | |
| `imull $16,(%eax,%edx,4)` | | |
| `incl 8(%eax)` | | |
| `decl %ecx` | | |
| `subl %edx,%eax` | | |

**3.8.** Suponer que queremos generar código ensamblador para la siguiente función C:

```
int shift_left2_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
   return x;
}
```

El código que sigue es una porción del código ensamblador que realiza los desplazamientos y deja el valor final en el registro %eax. Dos instrucciones clave han sido omitidas. Los parámetros x y n están almacenados en las posiciones de memoria con desplazamientos 8 y 12, respectivamente, relativos a la dirección almacenada en el registro %ebp.

```
1        movl 8(%ebp), %eax      Tomar x
2        _____________           x <<= 2
3        movl 12(%ebp), %ecx     Tomar n
4        _____________           x >>= n
```

Rellenar las instrucciones que faltan, siguiendo las anotaciones a la derecha. El desplazamiento a derecha debería realizarse aritméticamente.

**3.9.** En la siguiente variante de la función de la Figura 3.8(a) (similar a Tema2.2, tr.10), las expresiones se han dejado en blanco:

```
1    int arith(int x,
2              int y,
3              int z)
4    {
5        int t1 = __________;
6        int t2 = __________;
7        int t3 = __________;
8        int t4 = __________;
9        return t4;
10   }
```

La porción del código ensamblador generado que implementa estas expresiones es como sigue:

```
  x en %ebp+8, y en %ebp+12, z en %ebp+16
1        movl 12(%ebp), %eax
2        xorl 8(%ebp), %eax
3        sarl $3, %eax
4        notl %eax
5        subl 16(%ebp), %eax
```

Basándose en este código ensamblador, rellenar las partes que faltan en el código C.

**3.10.** Es común encontrar líneas de código ensamblador de la forma

```
xorl %edx,%edx
```

en código generado desde C en donde no había ninguna operación EXCLUSIVE-OR.

A. Explicar el efecto de esta particular instrucción EXCLUSIVE-OR y qué operación útil implementa.
B. ¿Cuál sería la forma más directa de expresar esta operación en código ensamblador?
C. Comparar el número de bytes para codificar estas dos diferentes formas de implementar la misma operación.

**3.11.** Modificar el código ensamblador mostrado para división con signo de forma que calcule el cociente y resto sin signo de los números `x` e `y` y almacene los resultados en la pila.

(El "código mostrado" se refiere a la página 218 del libro:

```
  x en %ebp+8, y en %ebp+12
1       movl 8(%ebp), %eax      Cargar x en %eax
2       cltd                    Extender signo en %edx
3       idivl 12(%ebp)          Dividir x (64b) por y
4       movl %eax, 4(%esp)      Guardar x / y
5       movl %edx, (%esp)       Guardar x % y
```

en donde también se dice qué cambio hay que hacer)

**3.12.** Considerar el siguiente prototipo de función C, donde `num_t` es un tipo de datos declarado usando `typedef`:

```
void store_prod(num_t *dest, unsigned x, num_t y) {
    *dest = x*y;
}
```

GCC genera el siguiente código ensamblador implementando el cuerpo del cálculo:

```
  dest en %ebp+8, x en %ebp+12, y en %ebp+16
1       movl 12(%ebp), %eax
2       movl 20(%ebp), %ecx
3       imull %eax, %ecx
4       mull 16(%ebp)
5       leal (%ecx,%edx), %edx
6       movl 8(%ebp), %ecx
7       movl %eax, (%ecx)
8       movl %edx, 4(%ecx)
```

Observar que éste código requiere dos lecturas de memoria para captar el argumento `y` (líneas 2 y 4), dos multiplicaciones (líneas 3 y 4), y dos escrituras a memoria para almacenar el resultado (líneas 7 y 8).

A. ¿Qué tipo de datos es `num_t`?
B. Describir el algoritmo usado para calcular el producto y argumentar que es correcto.

**3.13.** El siguiente código C

```
int comp(data_t a, data_t b) {
   return a COMP b;
}
```

muestra una comparación general entre los argumentos `a` y `b`, donde podemos ajustar el tipo de datos de los argumentos declarando `data_t` con una declaración `typedef`, y ajustar la comparación definiendo `COMP` con una declaración `#define`. Suponer que `a` está en `%edx` y `b` está en `%eax`. Para cada una de las siguientes secuencias de instrucciones, determinar qué tipos de datos `data_t` y qué

comparaciones COMP podrían causar que el compilador generara este código. (Puede haber múltiples respuestas correctas; se deben listar todas ellas)

A.
```
cmpl  %eax, %edx
setl  %al
```
B.
```
cmpw  %ax, %dx
setge %al
```
C.
```
cmpb  %al, %dl
setb  %al
```
D.
```
cmpl  %eax, %edx
setne %al
```

**3.14.** El siguiente código C

```
int test(data_t a) {
   return a TEST 0;
}
```

muestra una comparación general entre el argumento `a` y `0`, donde podemos ajustar el tipo de datos del argumentos declarando `data_t` con una declaración `typedef`, y la naturaleza de la comparación declarando TEST con una declaración `#define`. Para cada una de las siguientes secuencias de instrucciones, determinar qué tipos de datos `data_t` y qué comparaciones TEST podrían causar que el compilador generara este código. (Puede haber múltiples respuestas correctas; listar todas las correctas)

A.
```
testl %eax, %eax
setne %al
```
B.
```
testw %ax, %ax
sete  %al
```
C.
```
testb %al, %al
setg  %al
```
D.
```
testw %ax, %ax
seta  %al
```

**3.15.** En los siguientes extractos de un binario desensamblado, alguna información ha sido reemplazada por Xs. Responder las siguientes preguntas sobre estas instrucciones.

A. ¿Cuál es el destino de la instrucción `je` de abajo? (no se necesita saber nada sobre la instrucción `call` aquí)
```
804828f:    74 05                je XXXXXXX
8048291:    e8 1e 00 00 00       call 80482b4
```
B. ¿Cuál es el destino de la instrucción `jb` de abajo?
```
8048357:    72 e7                jb XXXXXXX
8048359:    c6 05 10 a0 04 08 01 movb $0x1,0x804a010
```
C. ¿Cuál es la dirección de la instrucción `mov`?
```
XXXXXXX:    74 12                je 8048391
XXXXXXX:    b8 00 00 00 00       mov $0x0,%eax
```
D. En el código que sigue, el destino de salto está codificado en forma relativa-al-PC como un número de 4-byte en complemento a 2. Los bytes se listan de menos a

más significativos, reflejando el ordenamiento de bytes little-endian de IA32. ¿Cuál es la dirección de destino del salto?

```
80482bf:    e9 e0 ff ff ff        jmp XXXXXXX
80482c4:    90                    nop
```

E. Explicar la relación entre la anotación a la derecha y la codificación en bytes a la izquierda.

```
80482aa:    ff 25 fc 9f 04 08     jmp *0x8049ffc
```

**3.16.** Cuando se le da el código C

```
void cond(int a, int *p)
{
    if (p && a > 0)
        *p += a ;
}
```

GCC genera el siguiente código ensamblador para el cuerpo de la función:

```
  a en %ebp+8, p en %ebp+12
1       movl 8(%ebp), %edx
2       movl 12(%ebp), %eax
3       testl %eax, %eax
4       je .L3
5       testl %edx, %edx
6       jle .L3
7       addl %edx, (%eax)
8     .L3:
```

A. Escribir una versión goto en C que realice el mismo cálculo e imite el flujo de control del código ensamblador, en el estilo mostrado en la Figura 3.13(b) (equivale a Tema2.2, tr.29). Podría resultar útil anotar primero el código ensamblador como hemos hecho en nuestros ejemplos.

B. Explicar por qué el código ensamblador contiene dos saltos condicionales, incluso aunque el código C tiene sólo una sentencia `if`.

**3.17.** Una regla alternativa para traducir sentencias `if` a código goto es como sigue:

```
    t = test-expr;
    if (t)
        goto true;
    else-statement
    goto done;
true:
    then-statement
done:
```

A. Reescribir la versión goto de `absdiff` (Figura 3.13(a-b) p.228, equivalente a Tema2.2, tr.28-32) basándose en esta regla alternativa.

B. ¿Se nos puede ocurrir alguna razón para preferir una regla a la otra?

**3.18.** A partir de código C de la forma

```
1    int test(int x, int y) {
2        int val = __________;
3        if (__________) {
4            if (__________) {
5                val = __________;
6            else
7                val = __________;
8        } else if (__________)
9            val = __________;
10       return val;
11   }
```

GCC genera el siguiente código ensamblador:

```
  x en %ebp+8, y en %ebp+12
1       movl   8(%ebp), %eax
2       movl   12(%ebp), %edx
3       cmpl   $-3, %eax
4       jge    .L2
5       cmpl   %edx, %eax
6       jle    .L3
7       imull  %edx, %eax
8       jmp    .L4
9    .L3:
10      leal   (%edx,%eax), %eax
11      jmp    .L4
12   .L2:
13      cmpl   $2, %eax
14      jg     .L5
15      xorl   %edx, %eax
16      jmp    .L4
17   .L5:
18      subl   %edx, %eax
19   .L4:
```

Rellenar las expresiones que faltan en el código C. Para conseguir que el código se corresponda con el patrón del código C, se tendrán que deshacer algunas de las reordenaciones de cálculos realizadas por GCC.

**3.19.**

A. ¿Cuál es el máximo valor de *n* para el cual podemos representar *n!* con un `int` de 32-bit?

B. ¿Qué sucede con un `long long int` de 64-bit?

**3.20.** Para el código C

```
1    int dw_loop(int x, int y, int n) {
2        do {
3            x += n;
4            y *= n;
5            n--;
```

```
6         } while ((n > 0) && (y < n));
7         return x;
8     }
```

GCC genera el siguiente código ensamblador:

```
  x en %ebp+8, y en %ebp+12, n en %ebp+16
1         movl   8(%ebp), %eax
2         movl   12(%ebp), %ecx
3         movl   16(%ebp), %edx
4      .L2
5         addl   %edx, %eax
6         imull  %edx, %ecx
7         subl   $1, %edx
8         testl  %edx, %edx
9         jle    .L5
10        cmpl   %edx, %ecx
11        jl     .L2
12     .L5:
```

A. Hacer una tabla con el uso de registros, similar a la mostrada en la Figura 3.14(b) (equivalente a Tema2.2, tr.35,39).
B. Identificar *test-expr* y *body-statement* en el código C, y las correspondientes líneas en el código ensamblador (Tema2.2, tr.40).
C. Añadir anotaciones al código ensamblador describiendo la operación del programa, similares a las mostradas en la Figura 3.14(b) (Tema2.2, tr.39).

**3.21.** Para el código C

```
1     int loop_while(int a, int b)
2     {
3         int result = 1;
4         while (a < b) {
5             result *= (a+b);
6             a++;
          }
7         return result;
8     }
```

GCC genera el siguiente código ensamblador:

```
  a en %ebp+8, b en %ebp+12
1         movl   8(%ebp), %ecx
2         movl   12(%ebp), %ebx
3         movl   $1, %eax
4         cmpl   %ebx, %ecx
5         jge    .L11
6         leal   (%ebx,%ecx), %edx
7      .L12
8         imull   %edx, %eax
9         addl   $1, %ecx
```

```
10        addl   $1, %edx
11        cmpl   %ecx, %ebx
12        jg     .L12
13      .L11
```

Al generar este código, GCC hace una interesante transformación que, en efecto, introduce una nueva variable del programa.

A. El registro `%edx` se inicializa en la línea 6 y se actualiza dentro del bucle en la línea 10. Considerar que esto es una nueva variable del programa. Describir qué tiene que ver con las variables en el código C.
B. Crear una tabla de uso de registros para esta función.
C. Anotar el código ensamblador para describir cómo opera.
D. Escribir una versión goto de la función (en C) que imite cómo opera el programa en código ensamblador.

**3.22.** Una función, `fun_a`, tiene la siguiente estructura general:

```
int fun_a(unsigned x) {
    int val = 0;
    while ( ________ ) {
        ________;
    }
    return ________;
}
```

El compilador GCC genera el siguiente código ensamblador:

```
  x en %ebp+8
1       movl   8(%ebp), %edx
2       movl   $0, %eax
3       testl  %edx, %edx
4       je     .L7
5     .L10:
6       xorl   %edx, %eax
7       shrl   %edx                Desplazar a derecha 1
8       jne    .L10
9     .L7:
10      andl   $1, %eax
```

Hacer ingeniería inversa de la operación de este código y entonces hacer lo siguiente:

A. Usar la versión en código ensamblador para rellenar las partes que faltan del código C.
B. Describir en castellano qué calcula esta función.

**3.23.** Una función, `fun_b`, tiene la siguiente estructura general:

```
int fun_b(unsigned x) {
    int val = 0;
    int i;
    for ( ________; ________; ________ ) {
        ________;
```

```
    }
    return val;
}
```

El compilador GCC genera el siguiente código ensamblador:

```
  x en %ebp+8
1       movl    8(%ebp), %ebx
2       movl    $0, %eax
3       movl    $0, %ecx
4     .L13:
5       leal    (%eax,%eax), %edx
6       movl    %ebx, %eax
7       andl    $1, %eax
8       orl     %edx, %eax
9       shrl    %ebx                    Desplazar a derecha 1
10      addl    $1, %ecx
11      cmpl    $32, %ecx
12      jne     .L13
```

Hacer ingeniería inversa de la operación de este código y entonces hacer lo siguiente:

A. Usar la versión en código ensamblador para rellenar las partes que faltan del código C.
B. Describir en castellano qué calcula esta función.

**3.24.** Ejecutar una sentencia `continue` en C causa que el programa salte al final de la iteración actual del bucle. La regla fijada para traducir un bucle `for` a un bucle `while` (Tema2.2, tr.45) necesita algo de elaboración para tratar con sentencias `continue`. Por ejemplo, considerar el siguiente código:

```
/* Ejemplo de bucle for usando una sentencia continue */
/* Suma los números pares entre 0 y 9 */
int sum = 0;
int i;
for ( i = 0; i < 10; i++ ) {
    if (i & 1)
        continue;
    sum += i;
}
```

A. ¿Qué obtendríamos si inocentemente hubiéramos aplicado nuestra regla para traducir el bucle `for` a un bucle `while`? ¿Qué estaría mal en dicho código?
B. ¿Cómo se podría sustituir la sentencia `continue` por una sentencia `goto` para asegurarse de que el bucle `while` duplica correctamente el comportamiento del bucle `for`?

**3.25.** Ejecutándose en un Pentium 4, nuestro código requirió alrededor de 16 ciclos cuando el patrón de saltos era altamente predecible, y alrededor de 31 ciclos cuando el patrón fue aleatorio.

A. ¿Cuál es la penalización por predicción errónea (*miss penalty*) aproximada?

B. ¿Cuántos ciclos requeriría la función cuando el salto se predice erróneamente?

La pregunta se refiere a la función `absdiff` con y sin movimiento condicional de la Figura 3.16 (equivalente a Tema2.2, tr.28-35), junto con la explicación de la página 242 en el 2º párrafo, los tiempos de ejecución en el párrafo 3º pp.242-243, y la explicación adicional (*"Aside"*) p.243, en donde se recuerda que los saltos condicionales vacían el pipeline si no se cumplen (o si se predicen incorrectamente).
En el párrafo 2º se menciona que se pueden perder 20-40 ciclos de reloj por predicción errónea. En el párrafo 3º que para el caso de `absdiff` en un Core i7 la función tarda 13 ciclos cuando acierta mucho y 35 ciclos cuando no puede predecir bien (50% de fallos), de donde se deduce que la penalización por fallar es de alrededor de 44 ciclos, y que cuando acierta tarda 13 ciclos y cuando falla 57 ciclos. En el *Aside* se explica cómo se calculan esos resultados. Este problema pretende comprobar si se entiende el método.
Por cierto, el código con movimiento condicional tardaba siempre 14 ciclos en el Core i7.

**3.26.** En la siguiente función C, hemos dejado la definición de la operación OP incompleta:

```
#define OP ________ /* Operador desconocido */

int arith(int x){
    return x OP 4;
}
```

Cuando se compila, GCC genera el siguiente código ensamblador:

```
  Registro: x en %edx
1     leal   3(%edx), %eax
2     testl  %edx, %edx
3     cmovns %edx, %eax
4     sarl   $2, %eax          Retornar valor en %eax
```

A. ¿Qué operación es OP?
B. Anotar el código para explicar cómo funciona.

**3.27.** A partir de código C de la forma

```
1   int test(int x, int y) {
2       int val = ________;
3       if (________) {
4           if (________)
5               val = ________;
6           else
7               val = ________;
8       } else if (________)
9           val = ________;
10      return val;
11  }
```

GCC, con el ajuste de línea de comandos '`-march=i686`', genera el siguiente código ensamblador:

```
  x en %ebp+8, y en %ebp+12
1         movl   8(%ebp), %ebx
2         movl   12(%ebp), %ecx
3         testl  %ecx, %ecx
4         jle    .L2
5         movl   %ebx, %edx
6         subl   %ecx, %edx
7         movl   %ecx, %eax
8         xorl   %ebx, %eax
9         compl  %ecx, %ebx
10        cmovl  %edx, %eax
11        jmp    .L4
12    .L2:
13        leal   0(,%ebx,4), %edx
14        leal   (%ecx,%ebx), %eax
15        cmpl   $-2, %ecx
16        cmovge %edx, %eax
17    .L4:
```

Rellenar las expresiones que faltan en el código C.

---

**3.48.** Una función C `arithprob` con argumentos a, b, c, y d tiene el siguiente cuerpo:

```
return a*b + c*d;
```

Compila al siguiente código X86-64:

```
1       arithprob
2         movslq %ecx, %rcx
3         imulq  %rdx, %rcx
4         movsbl %sil, %esi
5         imull  %edi, %esi
6         movslq %esi, %rsi
7         leaq   (%rcx,%rsi), %rax
8         ret
```

Los argumentos y valor de retorno son todos enteros con signo de diversas longitudes. Los argumentos a, b, c, y d se pasan en las regiones apropiadas de losregistros `%rdi`, `%rsi`, `%rdx`, y `%rcx`, respectivamente. Basándose en este código ensamblador, escribir un prototipo de función describiendo los tipos de los argumentos y de retorno para `arithprob`.

**3.49.** Una función `fun_c` tiene la siguiente estructura general:

```
long fun_c(unsigned long x) {
   long val = 0;
    int i;
```

```
    for ( ________ ; ________ ; ________ ) {
        ________;
    }
    ________;
    return ________;
}
```

El compilador C GCC genera el siguiente código ensamblador:

```
    x en %rdi
1    fun_c:
2      movl    $0, %ecx
3      movl    $0, %edx
4      movabsq $72340172838076673, %rsi
5    .L2:
6      movq    %rdi, %rax
7      andq    %rsi, %rax
8      addq    %rax, %rcx
9      shrq    %rdi                        Desplazar a derecha 1
10     addl    $1, %edx
11     cmpl    $8, %edx
12     jne     .L2
13     movq    %rcx, %rax
14     sarq    $32, %rax
15     addq    %rcx, %rax
16     movq    %rax, %rdx
17     sarq    $16, %rdx
18     addq    %rax, %rdx
19     movq    %rdx, %rax
20     sarq    $8, %rax
21     addq    %rdx, %rax
22     andl    $256, %eax
23     ret
```

Hacer ingeniería inversa de la operación de este código. Resultará útil convertir la constante decimal de la línea 4 a hexadecimal.

A. Usar la versión en código ensamblador para rellenar las partes que faltan del código C.
B. Describir en castellano qué calcula esta función.