Aritmética de enteros

Índice

1. Operadores y operaciones

PROBLEMA 1 La Figura 1 muestra tres operadores de suma, que se describen de la manera siguiente:

1. El sumador A es combinacional y tiene un tiempo de retardo de 10 ns.
2. El operador B es secuencial y se sincroniza con un reloj que oscila a 400 Mz. Para hacer una suma necesita 2 ciclos.
3. El operador C es también secuencial y va gobernado por un reloj que funciona a 150 MHz. En cada ciclo puede hacer dos sumas.

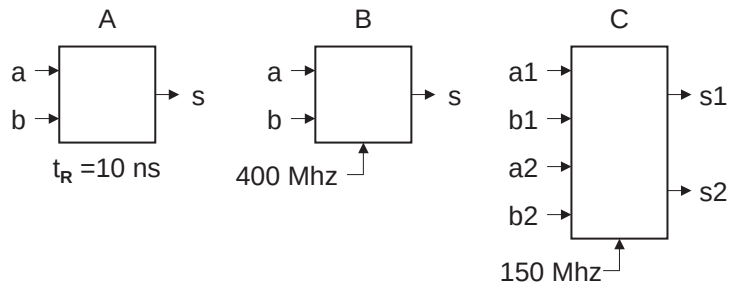

Figura 1: Tres operadors de suma. A es combinacional, B y C son secuenciales.

Calcule en MOPS cuál es la productividad máxima de cada operador.

SOLUCIÓN

1. El operador A es combinacional, con un retardo $t_R = 10$ ns. Si en la definición de productividad consideramos que la productividad máxima se obtendrá haciendo una operación cada t_R, escribiremos:

$$\text{Productividad máxima} = \frac{1 \text{ operación}}{10 \cdot 10^{-9} \text{ segundos}} = 100 \cdot 10^6 \text{ operaciones/segundo} = 100 \text{ MOPS}$$

Dicho de otra manera, el número de operaciones por segundo será la inversa del retardo del circuito.

2. El operador B es secuencial, y cambia de estado a una frecuencia de 400 MHz. Como dos ciclos de reloj son el mínimo tiempo para hacer una suma, podemos adaptar la definición de la productividad a los datos disponibles y escribir:

$$\text{Productividad máxima} = \frac{400 \cdot 10^6 \text{ ciclos/segundo}}{2 \text{ ciclos/operación}} = 200 \cdot 10^6 \text{ operaciones/segundo} = 200 \text{ MOPS}$$

   En este caso, la productividad es la frecuencia de reloj dividida por el número de ciclos por operación.

3. El operador C puede hacer como máximo dos sumas por ciclo. El cálculo es ahora

$$\begin{aligned} \text{Productividad máxima} &= 2 \text{ operaciones/ciclo} \times 150 \cdot 10^6 \text{ ciclos/segundo} \\ &= 300 \cdot 10^6 \text{ operaciones/segundo} = 300 \text{ MOPS} \end{aligned}$$

   En consecuencia, la productividad obtenida es el producto de la frecuencia del reloj por el número de operaciones por ciclo.

■

PROBLEMA 2 Os han encargado que diseñéis un operador de suma para un procesador que trabaja con una frecuencia de reloj de 500 MHz. Por las características del procesador, el operador ha de hacer una suma en cada ciclo de reloj, pero sólo dispone de la mitad del periodo de reloj para hacerla. ¿Cuál será el máximo retardo posible del operador? ¿Cuál será la productividad máxima (en MOPS) del operador?

SOLUCIÓN El periodo de reloj del procesador se puede calcular por medio de la expresión:

$$\text{periodo} = \frac{1}{500 \cdot 10^6 \text{ Hz}} = 2 \cdot 10^{-9} \text{ s} = 2 \text{ ns}$$

Así pues, el tiempo de respuesta del operador no podrá superar 1 ns.
La productividad del operador será el número de operaciones hechas por segundo. En este caso,

$$\text{Productividad} = 1 \text{ operación/ciclo} \times 500 \cdot 10^6 \text{ ciclos/segundo} = 500 \cdot 10^6 \text{ operaciones/segundo} = 500 \text{ MOPS}$$

■

PROBLEMA 3 Considere el código siguiente escrito en lenguaje ensamblador del MIPS R2000 que opera sobre el contenido del registro `$t1` y deja el resultado en el registro `$t0`.

```
        or $t0, $zero, $t1
        bgez $t1, etiq
        sub $t0, $zero, $t1
etiq:   ...
```

1. Si los números están representados en complemento a dos, ¿de qué operación se trata? Justifique la respuesta.

2. ¿Cuál sería el código alternativo si los números estuviesen expresados en signo y magnitud?

SOLUCIÓN

1. El código escribe en `$t0` el valor absoluto del número contenido en `$t1`. En concreto, primero copia `$t1` en `$t0`. Si el resultado ha sido positivo o igual a cero, entonces salta a la etiqueta y acaba el proceso; en cambio, si `$t1` es negativo, le cambia el signo (esto es, lo hace positivo) y lo copia en `$t0`.

2. Si los números se expresan en signo y magnitud será suficiente con poner a cero el bit de signo del valor que hay en el `$t1` y dejar el resultado en el registro `$t0`:

```
li $t0, 0x7FFFFFFF
and $t0, $t0, $t1
```

■

---

2. Suma, resta y comparación en el MIPS R2000

PROBLEMA 4 Escriba en ensamblador del MIPS R2000 el código de una función que retorne el mayor de sus dos parámetros enteros. Se debe seguir el convenio de programación usual: los parámetros se encuentran en `$a0` y `$a1` y la función ha de retornar el resultado en `$v0`. Hay que hacer dos versiones:

1. Una función `maxu` que opere con valores sin signo, es decir, que interprete los parámetros en CBN.
2. Una función `max` que opere con valores con signo, es decir, que interprete los parámetros en Ca2.

SOLUCIÓN Este ejercicio pide comparar los argumentos de la función atendiendo a los tipos de datos. Las dos soluciones son casi idénticas, sólo las instrucciones de comparación son diferentes porque dependen del tipo de operandos a comparar. Nótese que las dos soluciones retornan `$a1` cuando ambos argumentos son iguales.

1. Con valores sin signo, utilizaremos `sltu` para comparar los argumentos:

```
maxu:   sltu $t0,$a0,$a1
        beq $t0,$zero,es_a0
es_a1:  move $v0,$a1
        jr $ra
es_a0:  move $v0,$a0
        jr $ra
```

2. Si se trata de valores con signo, utilizaremos la función `slt` para comparar los argumentos:

```
max:    slt $t0,$a0,$a1
        beq $t0,$zero,es_a0
es_a1:  move $v0,$a1
        jr $ra
es_a0:  move $v0,$a0
        jr $ra
```

■

PROBLEMA 5 Queremos realizar operaciones de sumas y restas en el MIPS R2000, sin signo (CBN) y con signo (Ca2), pero detectando los desbordamientos producidos en cada operación (sin provocar nunca ninguna excepción) y, en su caso, saltar a la etiqueta `TratarOV` donde está la parte del código que los trata.

Escriba en ensamblador el código que implemente las sentencias de alto nivel siguientes, donde los símbolos hacen referencia a variables de 32 bits. Las variables a, b y c no tienen signo y d, e y f sí lo tienen.

1. $a = d$
2. $d = a$
3. $a = b + c$
4. $a = b - c$
5. $d = e + f$
6. $d = e - f$

SOLUCIÓN Las variables a,b y c son de 32 bits sin signo. Las variables d, e y f son de 32 bits con signo. En las funciones que hemos de escribir no pueden aparecer las instrucciones `add` y `sub` porque, en caso de desbordamiento aritmético, generarán una excepción OVF en el sistema. En su lugar emplearemos las instrucciones `addu` y `subu`, que realizan la misma operación pero no producen ninguna excepción. El desbordamiento lo detectaremos mediante el código apropiado en cada caso y, si se produce, llamaremos a la función `TratarOV`.

1. Para hacer $a = d$ no hece falta ninguna operación de suma o resta, sólo se trata de una asignación. Sin embargo, puede producirse una desbordamiento si la variable con signo d es negativa. El código apropiado es el siguiente:

```
lw $t0,d
bltz $t0,Tratar_OV
sw $t0,a
```

   El código anterior es equivalente a este otro:

```
lw $t0,d
slt $t1,$t0,$zero
bne $t1,$zero,Tratar_OV
sw $t0,a
```

   Una tercera alternativa podría consistir en analizar el bit de signo mediante operaciones lógicas con máscaras, como hace el código siguiente:

```
lw $t0,d
li $t1,0x80000000
and $t1,$t1,$t0
bne $t1,$zero,Tratar_OV
sw $t0,a
```

2. La operación $d = a$ puede producir desbordamiento si la variable sin signo a tiene un valor mayor que el máximo positivo $7FFFFFFF_{16}$ codificable mediante la variable d con signo. El código siguiente comprueba esta posibilidad y verifica que $d < 80000000_{16}$:

```
lw $t0,a
li $t1,0x80000000
sltu $t2,$t0,$t1
beq $t2,$zero,Tratar_OV
sw $t0,d
```

   Se puede observar que las soluciones alternativas del apartado anterior también pueden utilizarse en este caso.

3. El desbordamiento propio de una suma sin signo como $a = b + c$ se produce cuando el resultado es menor que cualquiera de los dos sumandos. Es decir, el resultado de la suma $s = b + c$, calculado por el procesador, será correcto si (y solo si) $s \geq b$. Podemos escribir una primera versión que detecte el desbordamiento comprobando la condición contraria $s < b$ con la instrucción `sltu`, que compara valores sin signo. Si falla la condición, el código asigna $a = s$, donde s es el registro `$t2`.

```
lw $t0,b
lw $t1,c
addu $t2,$t0,$t1
sltu $t3,$t2,$t0
bne $t3,$zero,Tratar_OV
sw $t2,a
```

   Igualmente, la suma será correcta si (y solo si) $s \geq c$. Tenemos entonces la solución alternativa:

```
lw $t0,b
lw $t1,c
addu $t2,$t0,$t1
sltu $t3,$t2,$t1
bne $t3,$zero,Tratar_OV
sw $t2,a
```

4. En el caso de la resta $a = b - c$, sabremos que la operación $r = b - c$, hecha por el procesador, es correcta si (y solo si) $r \leq b$. Igualmente, la resta es correcta si (y solo si) $b \geq c$. Nótese que estas condiciones son equivalentes a las vistas en el caso anterior de la suma, pero considerando que $r = b - c \Rightarrow b = r + c$.

   En consecuencia, tenemos dos soluciones alternativas. La primera comprueba que $r \leq b$ antes de asignar el resultado a la variable a:

```
lw $t0,b
lw $t1,c
subu $t2,$t0,$t1
sltu $t3,$t0,$t2
bne $t3,$zero,Tratar_OV
sw $t2,a
```

   La segunda verifica que $b \geq c$ (o que $c < b$) antes de dar por bueno el resultado calculado.

```
lw $t0,b
lw $t1,c
subu $t2,$t0,$t1
sltu $t3,$t0,$t1
bne $t3,$zero,Tratar_OV
sw $t2,a
```

5. El caso de la suma con signo $d = e + f$ es más complicado. Cuando el procesador calcula $s = e + f$, sabremos que la operación ha producido un desbordamiento si los signos de los operandos son iguales y, además, distinto del signo del resultado.

   El código siguiente suma los dos valores mediante `addu`. Después utiliza la instrucción `xor` para comparar, bit a bit, los dos operandos. El unico bit que importa es el de signo. Si los dos operando tienen el mismo signo, la operación `xor` dará 0 en el bit de signo del resultado. En caso contrario dará un 1 (negativo).

   Con una instrucción `and` y la máscara correspondiente se puede aislar el bit de signo y, si es igual a 1, el resultado puede considerarse correcto (no hay desbordamiento). En caso contrario habrá que seguir el análisis, porque nos encontraremos en el caso de que los operandos tienen el mismo signo. En este caso, deberemos comparar el signo del resultado con el de cualquiera de los dos operandos, de nuevo empleando la instrucción `xor`.

```
        lw $t0,e
        lw $t1,f
        li $t4,0x80000000  # la máscara
        # primero hay que sumar
        addu $t2,$t0,$t1
        # ¿son iguales los signos de los operandos?
        xor $t3,$t0,$t1
        and $t3,$t3,$t4
        bne $t3,$zero,OK # si distintos, no hay problema
        # atención: ¿coinciden con el signo del resultado?
        xor $t3,$t0,$t2  # también podría ser xor $t3,$t1,$t2
        and $t3,$t3,$t4
        bne $t3,$zero,Tratar_OV
OK:     sw $t2,d
```

   Este código se puede simplificar con un truco de programación en ensamblador. Para bifurcar en función del bit más significativo de un registro, basta con utilizar las instrucciones `bltz` o `bgez` aunque su operando no tenga un significado numérico normal.

```
lw $t0,e
lw $t1,f
addu $t2,$t0,$t1
xor $t3,$t0,$t1
bltz $t3,OK
```

```
        xor $t3,$t0,$t2   # también podría ser xor $t3,$t1,$t2
        bltz $t3,Tratar_OV
OK:     sw $t2,d
```

6. Para tratar la resta de variables con signo $d = e - f$ hay que comprobar si el resultado calculado $r = e - f$ es correcto adaptando los criterios de la suma a este caso. El resultado r ha de satisfacer que $e = r + f$: si r y f tienen el mismo signo, éste ha de coincidir con el signo de e.

```
        lw $t0,e
        lw $t1,f
        subu $t2,$t0,$t1
        xor $t3,$t2,$t1  # para ver si r y f tienen distinto signo
        bltz $t3,OK
        xor $t3,$t1,$t0   # Atención, r y f tienen el mismo signo.
         # si el signo de cualquiera de ellos es distinto del de e
        bltz $t3,Tractar_OV  # entonces hay desbordamiento
OK:     sw $t2,var_d
```

■

PROBLEMA 6 En todos los sistemas de representación binaria con n bits se pueden encontrar un máximo valor representable M y un mínimo m. En un sistema de representación dado se define la suma con saturación ss a la operación definida así:

$$ss(x,y) = \begin{cases} m & \text{si } (x+y) < m \\ x+y & \text{si } m \leq (x+y) \leq M \\ M & \text{si } M < (x+y) \end{cases}$$

Es decir, la suma con saturación coincide con la suma convencional mientras el resultado se encuentre dentro del rango $[m \ldots M]$; en caso de desbordamiento el resultado es el mínimo o el máximo representable (el que esté más cerca del resultado no saturado). De la misma manera se puede definir la substracción con saturación. La mayoría de los procesadores actuales tienen instrucciones que implementan este tipo de aritmética imprescindible para los gráficos, pero el MIPS R2000 es demasiado antiguo para eso.

Tenemos que hacer una biblioteca de funciones, en lenguaje ensamblador del MIPS R2000, con las funciones siguientes:

1. `ss8u`: Suma con saturación de números de 8 bits sin signo.
2. `ss8`: Suma con saturación de números de 8 bits con signo.
3. `ss32u`: Suma con saturación de números de 32 bits sin signo.
4. `ss32`: Suma con saturación de números de 32 bits con signo.

Las funciones seguirán el convenio de paso de parámetros por los registros `$a0` y `$a1` y devolverán el resultado en `$v0`. En los dos primeros casos, las funciones podrán suponer que los parámetros recibidos están bien formados, es decir, que se encuentran dentro del rango de números representables con 8 bits y con los 24 bits restantes a 0 en caso de números sin signo, y con el signo extendido en el caso de número con signo. La Figura 2 muestra los formatos utilizados.

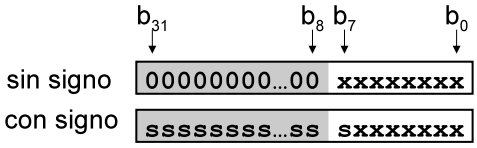

Figura 2: Formatos de 8 bits. Los bits expresados como `s` son todos iguales y representan el bit de signo.

SOLUCIÓN

1. ss8u: Suma con saturación de 8 bits sin signo.

   Para hacer la suma sólo tenemos las instrucciones convencionales add y addu. Podemos utilizar cualquiera de las dos porque no se puede dar una situación de desbordamiento ya que el resultado obtenido tiene 32 bits. Una vez hecha la suma $x + y$, la función deberá comprobar si el resultado no desborda, en el sentido de si está en el rango $0 \ldots 255$. Si los valores no tienen signo, sólo puede producirse desbordamiento si el resultado es mayor que 255, como se indica en la figura 3.

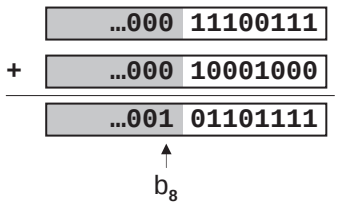

   Figura 3: Caso de desbordamiento en la aritmética con saturación de 8 bits sin signo.

   Para detectar la saturación (o desbordamiento) bastará con comparar el resultado con 255 mediante la instrucción sltu. Si la suma supera este valor, habrá que sustituir el resultado por 255; en caso contrario el resultado ya es correcto.

```
ss8u:     addu $v0,$a0,$a1
          sltiu $t0,$v0,256
          bne $t0,$zero,fin_ss8u
          li $v0,255
fin_ss8u: jr $ra
```

   La detección de la saturación también se puede realizar mediante el análisis del bit b_8 del resultado, usando instrucciones lógicas. Si $b_8 = 1$, se ha producido la saturación.

```
ss8u:     addu $v0,$a0,$a1
          andi $t0,$v0,0x100
          beq $t0,$zero,fin_ss8u
          li $v0,255
fin_ss8u: jr $ra
```

2. ss8: Suma con saturación de 8 bits con signo

   La aritmètica con signo puede desbordar en ambos sentidos: si x e y son ambos positivos, el resultado puede superar el máximo positivo $(x + y) > M$; si x e y son ambos negativos, el resultado puede ser menor que el mínimo negativo $(x + y) < m$.

   Para números de 8 bits tenemos $m = -128$ y $M = +127$. Para implementar ss8 se puede tratar cada caso por separado. La figura 4 muestra ejemplos de ambos casos.

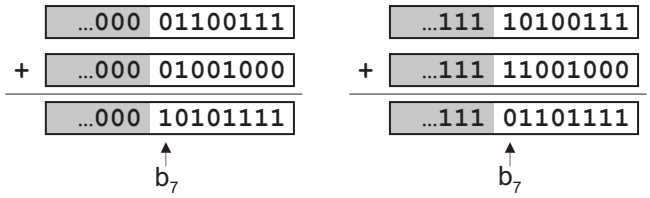

   Figura 4: Dos casos de saturación con signo.

   La detección del desbordamiento puede hacerse mediante dos comparaciones, una para cada extremo del rango, como en esta solución:

```
ss8:      addu $v0,$a0,$a1
porarriba:slti $t0,$v0,128
          bne $t0,$zero,porabajo
```

```
          li $v0,127
          j fi_ss8
porabajo: slti $t0,$v0,-128
          beq $t0,$zero,fin_ss8
          li $v0,-128
fin_ss8:  jr $ra
```

3. ss32u: Suma con saturación de 32 bits sin signo.

   El problema de suma con saturación para números de 32 bits sin signo es equivalente al de la detección del desbordamiento del ejercicio 5. La diferencia es que, en este caso, si el resultado desborda habrá que devover el valor límite $M = FFFFFFFF_{16}$.

```
ss32u:    addu $v0,$a0,$a1
          sltu $t0,$v0,$a0
          beq $t0,$zero,fin_ss32u
          li $v0,0xffffffff
fin_ss32u:jr $ra
```

4. ss32: Suma con saturación de 32 bits con signo.

   Para la suma con saturación de 32 bits con signo, podemos adaptar la solución del problema 5. La solución mostrada hace el mismo análisis de los signos para buscar el desbordamiento. Una vez detectado, se tratan por separado los dos casos posibles, que el resultado sea $(x + y) > M$ o que sea $(x + y) < m$. Para distinguirlos podemos analizar el bit de signo del resultado, considerando que el desbordamiento siempre invierte este bit. Si el resultado aparece como positivo deberemos considerar que el cálculo ha desbordado por el extremo inferior, es decir que $(x + y) < m$; en caso contrario sabremos que ha desbordado por el extremo superior y, por lo tanto, que $(x + y) > M$. Después habrá que ajustar el resultado a m en el primer caso y a M en el segundo.

```
ss32:      addu $v0,$a0,$a1
           # detección del desbordamiento
           xor $t0,$a0,$a1
           bltz $t0,fin_ss32
           xor $t0,$a0,$v0
           bgez $t0,fin_ss32
           # ¿por dónde ha desbordado?
           bgez $v0,porabajo
           # ajuste
porarriba: li $v0,0x7fffffff
           j fin_ss32
porabajo:  li $v0,0x80000000
fin_ss32:  jr $ra
```

■

PROBLEMA 7 Escriba en ensamblador un programa que sume dos variables *var_b* y *var_c* y asigne el resultado a *var_a*. Las tres variables son del tipo *long* de Java (entero de 64 bits). Como el MIPS R2000 no soporta este tipo de datos, fijaremos un convenio de uso antes de comenzar:

- Cada variable se ubicará en dos palabras consecutivas de memoria, que tendrán direcciones A y $A + 4$; la parte menos significativa de la variable se almacenará en A y la parte más significativa en $A + 4$.
- El convenio de comunicación con el programa llamante es el siguiente: el primer parámetro se encuentra en $a1||$a0; es decir, que los 32 bits más significativos se encuentran en $a1 y los menos significativos en $a0. El segundo argumento se encuentra en $a3||$a2. La función ha de devolver su resultado en $v1||$v0.

Aplicando el convenio descrito, siga los pasos siguientes para llegar a la solución completa:

1. Escriba en ensamblador una función `sumDPu` que calcule la suma de dos parámetros de 64 bits que no provoquen excepción en caso de desbordamiento.

2. Escriba en ensamblador la sección de datos del programa, donde han de estar ubicadas las tres variables *var_a*, *var_b* y *var_c*. Los valores iniciales han de ser:

Variable	Contenido (hex)
var_a	0000 0000 0000 0000
var_b	3333 5555 FFFF 0000
var_c	AAAA 2222 1111 7777

3. Complete el programa escribiendo el código ejecutable principal que lea de la memoria las variables *var_b* y *var_c*, llame `sumDPu` para sumarlas y escriba en *var_a* el resultado.

4. ¿Qué cambiaría de todo el programa que ha escrito en los apartados anteriores si las variables tuvieran signo y quisiera que los desbordamientos generaran una excepción OVF?

SOLUCIÓN

1. Diseño de la función `sumDPu`:

```
sumDPu:     addu $v0,$a0,$a2
            sltu $v1,$v0,$a0
            addu $v1,$v1,$a1
            addu $v1,$v1,$a3
            jr $ra
```

2. Especificación de las variables. Como el ensamblador del MIPS R2000 ubica los objetos descritos en el archivo fuente por orden creciente de direcciones, en el área de variables, a continuación de la etiqueta, habrá que especificar primero la palabra menos significativa y después la más significativa. Vea cómo se describen las variables de este problema:

```
.data
var_a:      .word 0x00000000,0x00000000
var_b:      .word 0xffff0000,0x33335555
var_c:      .word 0x11117777,0xaaaa2222
```

3. En el cuerpo del programa principal hay que cargar los registros que hacen el paso de parámetros a la función. Note como, para cada uno de ellos, la palabra menos significativa se obtiene mediante la etiqueta, mientras que la palabra más significativa es accesible con un desplazamiento de 4. Igualmente se trata el resultado de la función.

```
            # carga de var_b
            la $t0,var_b
            lw $a0,0($t0)
            lw $a1,4($t0)

            # carga de var_c
            la $t0,var_c
            lw $a2,0($t0)
            lw $a3,4($t0)

            # llamada a sumDP
            jal sumDPu

            # escritura de var_a
            la $t0,var_a
            sw $v0,0($t0)
            sw $v1,4($t0)
```

4. Para un tratamiento de la aritmética con signo equivalente al de la instrucción add, sólo habría que modificar la función. Vea el código de la función resultante sumDP: hay que substituir las dos instrucciones de suma que calculan la parte más significativa del resultado. El desbordamiento puede darse en dos momentos: cuando se suma la contribución del transporte al primer operando y cuando se suma el segundo operando. En ambos casos, la instrucción apropiada es add en lugar de addu.

```
sumDP:      addu $v0,$a0,$a2
            sltu $v1,$v0,$a0
            add $v1,$v1,$a1
            add $v1,$v1,$a3
            jr $ra
```

■

PROBLEMA 8 Escriba en ensamblador de MIPS R2000 el código de una función maxDP que devuelva el mayor de sus dos argumentos de tipo *long* de Java (entero de 64 bits). Siga el convenio de programación definido en el problema 7.

SOLUCIÓN Para el diseño de maxDP podemos tratar los argumentos comenzando por su parte más significativa con la instrucción sltu. La comparación se puede resolver enseguida si las partes más significativas de ambos operandos no son iguales. En el caso que sean iguales, habrá que comparar las partes menos significativas. Las dos comparaciones se llevan a cabo con la instrucción sltu.

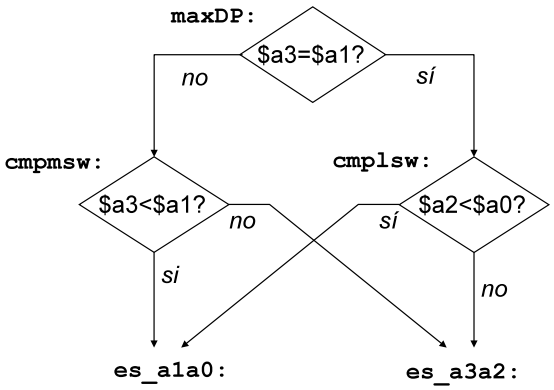

Figura 5: Árbol de decisiones de maxDP.

La Figura 5 ilustra el árbol de decisiones que guía el código de la solución que se muestra a continuación. En la etiqueta es_a3a2 está codificado el tratamiento aplicable si $a3||$a2>$a1||$a0 y en es_a3a2 el caso contrario. Note que si ambos argumentos son iguales, la función devuelve $a3||$a2.

```
maxDP:    beq $a1,$a3,cmplsw
   # caso con $a1 /= $a3
cmpmsw:   slt $t0,$a1,$a3
          beq $t0,$zero,es_a1a0
          j es_a3a2
   # caso con $a1 = $a3
cmplsw:   sltu $t0,$a0,$a2
          beq $t0,$zero,es_a1a0
   # tratamiento de $a3|$a2 >= $a1|$a0
es_a3a2:  move $v0,$a2
          move $v1,$a3
          jr $ra
   # tratamiento de $a3|$a2 < $a1|$a0
es_a1a0:  move $v0,$a0
          move $v1,$a1
          jr $ra
```

■

---

3. Diseño de operadores de suma y resta

PROBLEMA 9 Las Figuras 6 y 7 muestran dos implementaciones del sumador completo. El circuito de la Figura 6 se deduce directamente de las funciones lógicas y el de la Figura 7 combina dos semisumadores con una puerta *or*.

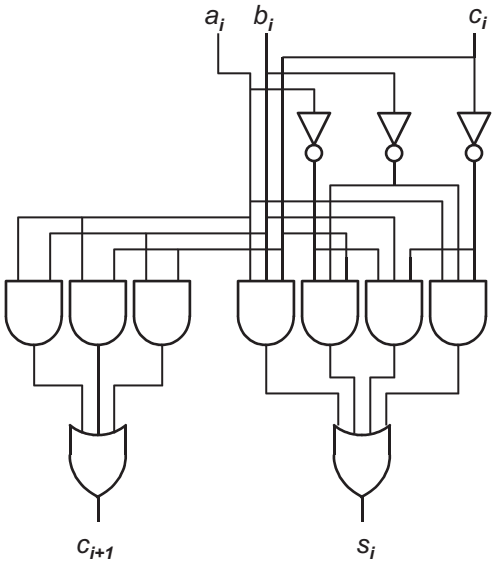

Figura 6: Implementación del sumador de un bit a partir de las funciones lógicas.

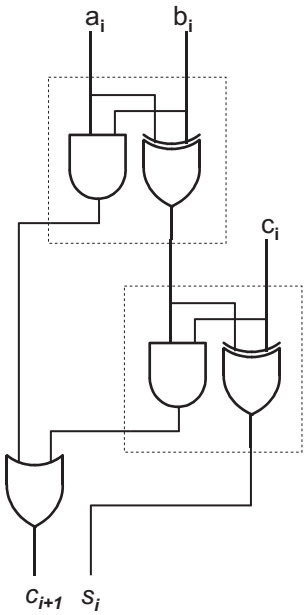

Figura 7: Implementación del sumador de un bit mediante semisumadores.

Considere los retardos de las puertas listadas en la tabla siguiente:

Puerta	Retardo (ns)
not	0,5
and y or de 2 entradas	1
exor de 2 entradas	2
and y or de 3 entradas	1,2
or de 4 entradas	1,5

Con estos datos, calcule los retardos de cada una de las salidas de los circuitos siguientes:

1. El sumador de un bit mostrado en la Figura 6.

2. El sumador de un bit mostrado en la Figura 7.
3. Un sumador de 4 bits construido con sumadores como los de la Figura 6 conectados en serie.
4. Sumador de 4 bits construido con sumadores como los de la Figura 7 conectados en serie.

SOLUCIÓN

1. Haremos el análisis del sumador completo implementado a partir de las funciones lógicas mostrado en la Figura 6. Este circuito tiene dos salidas: el bit de suma s_i y la salida de acarreo c_{i+1}.

   - **bit de suma s_i:** El nivel de esta salida es 3. El retardo es de $t_S = t_{Not} + t_{And(3)} + t_{Or(4)} = 0{,}5 + 1{,}2 + 1{,}5 = 3{,}2$ ns
   - **bit de acarreo c_{i+1}:** Esta salida es de nivel 2, y el retardo $t_C = t_{And(2)} + t_{Or(3)} = 1 + 1{,}2 = 2{,}2$ ns

   Note que, por simetría, estos retardos son iguales para todas las entradas. Es decir, un cambio en cualquier entrada a_i, b_i o c_i tendrá efecto en las salidas después de los retardos calculados.

2. En cuanto a la implementación mediante semisumadores, los retardos son:

   - **bit de suma s_i:** El nivel de esta salida es 2. En ambos niveles, formados por semisumadores (vea la Figura 7), la puerta *exor* determina el retardo. El retardo total es de $t_S = 2 \times t_{Exor} = 4$ ns.
   - **bit de acarreo c_{i+1}:** Esta salida es de nivel 3. En el primer nivel está el semisumador superior de la Figura 7 con un retardo de t_{Exor}, en el segundo nivel está el semisumador inferior con un retardo de $t_{And(2)}$ y en el tercero está la puerta *or*. Sumándolo todo, resulta $t_C = t_{Exor} + t_{And(2)} + t_{Or(2)} = 4$ ns.

   En esta implementación ha desaparecido la simetría entre las entradas. Un cambio en una entrada a_i o b_i tendrá efecto en las salidas s_i y c_{i+1} después de los retardos t_C y t_S calculados más arriba; pero un cambio en la entrada c_i, manteniendo las otras entradas estables, afectará a las salidas con unos retardos relativos menores. El retardo relativo de la salida s_i respecto de la entrada c_i lo expresaremos como $t_{C\to S} = t_{Exor} = 2$ ns; el retardo relativo de c_{i+1} es $t_{C\to C} = t_{And} + t_{Or} = 2$ ns. Estos retardos relativos son válidos $t_{Exor} = 2$ ns o más después de haberse estabilizado las entradas a_i o b_i.

3. Para el análisis temporal del sumador serie de 4 bits construido con sumadores implementados a partir de las funciones lógicas sólo hay que considerar los retardos t_S y t_C calculados en el apartado 1, como se muestra en la Figura 8.

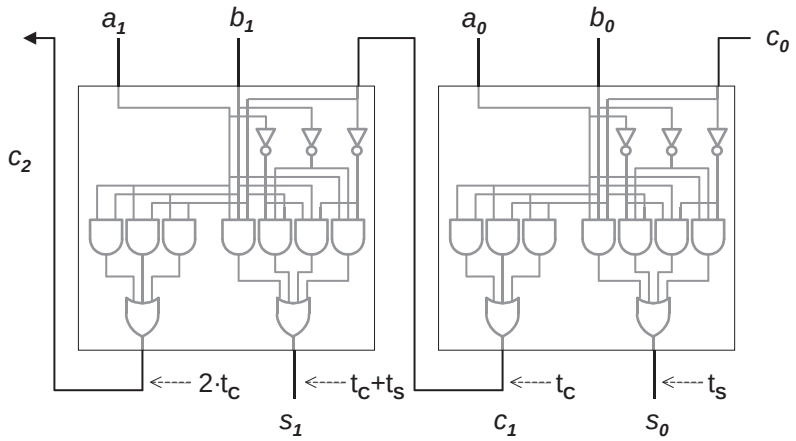

Figura 8: Las dos etapas menos significativas del sumador serie de 4 bits.

En general, el retardo de los bits de acarreo c_i es de $t_C \cdot i$ y el retardo de las salidas de suma s_i és $t_C \cdot i + t_S$. Aplicando los resultados del apartado 1, tenemos los retardos de les salidas, junto con los retardos de los acarreos intermedios, en la tabla siguiente:

Señal	Retardo (ns)	Señal	Retardo (ns)
s_0	3,2	c_1	2,2
s_1	5,4	c_2	4,4
s_2	7,6	c_3	6,6
s_3	9,8	c_4	8,8

4. Vea en la Figura 9 el análisis del sumador serie de 4 bits construido con semisumadores. Los retardos de la salida s_0 y del acarreo c_1 son los valores $t_S = 4$ ns y $t_C = 4$ ns calculados en el apartado 2. Para calcular los retardos de la salida s_1 y del acarreo c_2 a partir del momento en que c_1 toma su valor, hemos de entender que las entradas a_1 y b_1 llevan más de 2 ns estables y que por eso podemos aplicar los retardos $t_{C\to C}$ y $t_{C\to S}$ relativos al acarreo. Generalizando, podemos escribir el retardo de los bits de acarreo c_i $(i > 0)$ como $t_C + t_{C\to C} \cdot (i-1)$ y el retardo de las salidas de suma s_i $(i > 0)$ como $t_C + t_{C\to C} \cdot (i-1) + t_{C\to S}$.

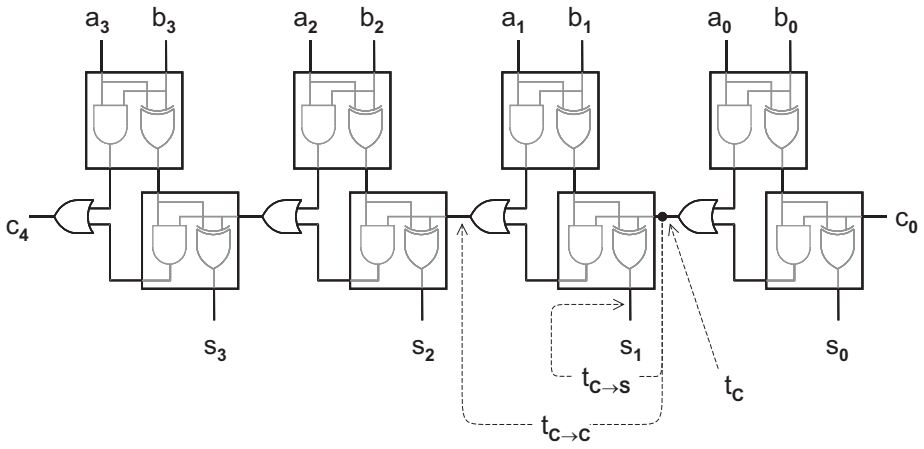

Figura 9: Sumador serie de 4 bits hecho con semisumadores.

Dando valores numéricos, obtenemos los retardos de las salidas y de los acarreos:

Señal	Retardo (ns)	Señal	Retardo (ns)
s_0	4	c_1	4
s_1	6	c_2	6
s_2	8	c_3	8
s_3	10	c_4	10

■

PROBLEMA 10 Hay una técnica de diseño de sumadores rápidos llamada *Carry Select Adder*, que adaptaremos a nuestros diseños. Se trata de estructurar el operador de n bits en secciones de m bits que trabajan en paralelo. En cada sección hay dos sumadores de m bits que calculan la suma de los bits implicados por duplicado: un operador asume que el acarreo de entrada a la sección es 0 y el otro que es 1. Cuando se ha determinado el valor real del acarreo de entrada a la sección, un multiplexor selecciona el resultado correcto. Vea en la Figura 10 un caso con $n = 32$ bits y $m = 16$ bits.

Calcule el retardo y la productividad del circuito de la Figura 10. Considere que las puertas tienen un retardo de 1 ns, que los sumadores están implementados a partir de las funciones lógicas y que el multiplexor es un circuito de nivel 3. Con esta disposición, utilice sumadores CPA de 16 bits.

SOLUCIÓN Dado que la salida de acarreo de un sumador completo tiene un retardo de $t_C = 2$ ns, el retardo de la salida c_{16} de un sumador CPA de $n = 16$ bits será $t_{op} = n \cdot t_C = 32$ ns; el retardo de las salidas de suma $s_{15} \ldots s_0$, como vimos en el problema **??**, es de 33 ns.

En este caso de sumador de 32 bits, habrá que considerar el retardo adicional de 3 ns a causa del multiplexor, que contaremos a partir del instante $t = 33$ ns en el que se le aplican todas las entradas. En total, el retardo es de 36 ns y la productividad resultante será de 27,8 MOPS.

■

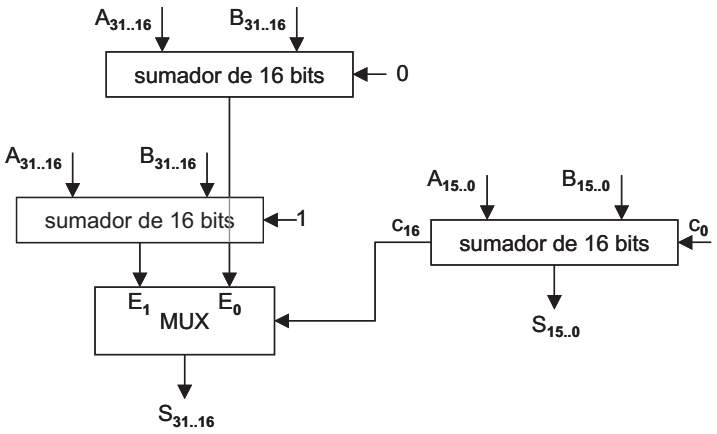

Figura 10: Aplicación de la técnica *Carry Select Adder* a un sumador de 32 bits.

4. Multiplicación con el MIPS R2000

PROBLEMA 11 Escriba en lenguaje ensamblador del MIPS R2000 una función con la especificación siguiente:

$$\texttt{mul2en: \$v0} = \texttt{\$a0} \times 2^{\texttt{\$a1}}$$

La función no puede hacer uso de las instrucciones de multiplicación del MIPS y sólo podrá utilizar instrucciones de suma, resta y desplazamiento. No ha de detectar desbordamientos aritméticos.

SOLUCIÓN En general, para multiplicar en binario por la potencia n-ésima de 2, hay que hacer un desplazamiento de n posiciones hacia la izquierda. Eso es precisamente lo que hace la instrucción MIPS `sllv`:

```
mul2en: sllv $v0,$a0,$a1
        jr $ra
```

■

PROBLEMA 12 Realice, en lenguaje ensamblador del MIPS R2000, una biblioteca de funciones con las especificaciones

1. `mul4`: $v0 = $a0 $\times$ 4
2. `mul5`: $v0 = $a0 $\times$ 5
3. `mul10`: $v0 = $a0 $\times$ 10
4. `mul15`: $v0 = $a0 $\times$ 15

Las rutinas no pueden hacer uso de las instrucciones de multiplicación del MIPS y sólo podrán utilizar instrucciones de suma, resta y desplazamiento. No hay que detectar el desbordamiento.

SOLUCIÓN

1. `mul4`: El método más directo para multiplicar por cuatro es el desplazamiento del operando hacia la izquierda en dos posiciones.

   ```
   mul4:   sll $v0,$a0,2
           jr $ra
   ```

2. `mul5`: Aquí hay que multiplicar `$a0` por una constante que no es potencia de dos. Podemos descomponer el factor 5 en suma de dos potencias de dos: $4 + 1$. Por la propiedad distributiva del producto respecto a la suma, tenemos que $\texttt{\$a0} \times 5 = \texttt{\$a0} \times (4 + 1) = \texttt{\$a0} \times (2^2 + 2^0)$, como se refleja en el código siguiente:

```
mul5:   sll $v0,$a0,2    # $v0 = 4*$a0
        addu $v0,$v0,$a0 # $v0 = 4*$a0 + $a0 = 5*$a0
        jr $ra
```

3. `mul10`: Para multiplicar por 10, podemos aprovechar la multiplicación por 5, hecha en el apartado 2, y multiplicar el resultado por 2. El resultado es la función siguiente:

```
mul10:  sll $v0,$a0,2
        addu $v0,$v0,$a0
        sll $v0,$v0,1
        jr $ra
```

   Si descomponemos 10 en suma de potencias de 2 tenemos $10 = 8 + 2 = 2^3 + 2^1$. Teniendo esto en cuenta podemos escribir una nueva versión de la función:

```
mul10:  sll $v0,$a0,3
        sll $t0,$a0,1
        addu $v0,$v0,$t0
        jr $ra
```

   Estas técnica de multiplicación de una variable por una constante, mediante desplazamientos y sumas, es muy cómoda cuando la constante tiene pocos bits a 1. Nótese que para una constante que tenga n bits a 1 habría que hacer $n - 1$ sumas y n desplazamientos. (Podemos ahorrar un desplazamiento si aparece 2^0 en la descomposición, como nos ha pasado en el apartado 2)

4. `mul15`: Una primera solución, basada en la descomposición de 15 en suma de potencias de 2: $15 = 2^3 + 2^2 + 2^1 + 2^0$ sería esta:

```
mul15:  sll $v0,$a0,3
        sll $t0,$a0,2
        addu $v0,$v0,$t0
        sll $t0,$a0,1
        addu $v0,$v0,$t0
        addu $v0,$v0,$a0
        jr $ra
```

   Notése que hacen falta 3 sumas y 3 desplazamientos, como corresponde a la descomposición del 15 en cuatro potencias de 2.

■

PROBLEMA 13 En el problema 12 se han diseñado funciones que multiplican valores sin signo sin detectar desbordamientos. ¿Cómo podrian detectar el desbordament estas funciones?

SOLUCIÓN Para detectar desbordamientos con operandos enteros sin signo no podemos adaptar la técnica aplicada en el problema 5 (apartado 3). No basta comparar el producto calculado con un factor `$a0`. Por ejemplo, si la función calcula $m = 4 \times$ `$a0`, cuando `$a0` $= 60000000_{16}$ resultaría $m = 80000000_{16}$, y se cumple que $m >$ `$a0` aunque la operación ha desbordado.

En un desplazamiento hacia la izquierda la verificación ha de hacerse comprobando que los bits eliminados son todos cero (recuerde que estamos trabajando con operandos sin signo). En el ejemplo de la multiplicación por 4, habrá que comprobar que los dos bits más significativos de `$a0` son cero, ya que `$a0` se tendrá que desplazar 2 posiciones a la izquierda para multiplicarlo por 4:

```
# código para multiplicar por 4 (valores sin signo)
        sll $v0,$a0,2
        # añadido para detectar desbordamientos
        li $t0,0xC0000000
        and $t0,$t0,$a0
        bne $t0,$zero,Tratar_Desbordamiento
```

En productos más complicados (por ejemplo, en una multiplicación por 10) habrá que verificar todos los desplazamientos que se hacen. También habrá que verificar las sumas y las restas con comparaciones como las del apartado 3 del problema 5.

Por último, conviene recordar que la comprobación de desbordamiento en la multiplicación no es necesaria si se reservan $2n$ bits para el producto.

■

PROBLEMA 14 Considere que está trabajando con un MIPS R2000 a una frecuencia de reloj de 100 MHz y que todas las instrucciones de enteros se ejecutan en un ciclo, excepto las instrucciones generales de multiplicación `mult` y `multu` que necesitan 6 ciclos.

1. ¿Cuál es el tiempo de ejecución de las funciones que hemos escrito en el problema 12?
2. ¿Cuál sería el tiempo de ejecución de las funciones si se hubieran utilizado las instrucciones generales de multiplicación?

Considere que no hace falta detectar desbordamiento y que los resultados de todos los productos caben en un registro de 32 bits.

SOLUCIÓN

1. Las funciones que hemos escrito en el problema 12 necesitan $10 \times n$ ns para ejecutarse, donde n es el número de instrucciones que contienen. Por ejemplo, la solución al apartado 4 (`mul15`), con 7 instrucciones, tiene un tiempo de ejecución de 70 ns.
2. Tomemos como ejemplo la multiplicación por 15 por medio de la instrucción `mul`. La implementación seria esta:

```
mul15:    li $t0,15
          mult $a0,$t0
          mflo $v0
          jr $ra
```

   El tiempo de ejecución seria ahora de $60 + 3 \times 10 = 90$ ns. Cualquier función que contenga menos de 9 instrucciones, contando entre sumas, restas, desplazamientos y el retorno final, será más rápida que esta última, como es el caso de todas las soluciones del problema 12.

   En sus recomendaciones para optimización de código, los diseñadores de los procesadores suelen aconsejar que los compiladores traduzcan las multiplicaciones por constantes pequeñas conocidas en el momento de la compilación mediante las técnicas que hemos practicado en el problema 12.

■

---

5. Operadores de multiplicación

PROBLEMA 15 Aplicando el algoritmo de multiplicación de números sin signo por medio de un operador secuencial, realice los productos siguientes en base 2 utilizando 5 bits:

1. 12×5
2. 18×11
3. 25×19
4. 30×22

SOLUCIÓN Para todos los casos, en primer lugar se ha de obtener la codificación binaria de cada uno de los operandos. En el paso de inicialización, el registro M se carga con el valor del multiplicando, el multiplicador se almacena en el registro LO y el bit C se inicializa a cero.

En cada paso del algoritmo se comienza por analizar el bit de menor peso del multiplicador (LO_0) para determinar si se ha de sumar M a HI (si $LO_0 = 1$) o no (si $LO_0 = 0$). Al final de cada paso, siempre se ha de desplazar el valor contenido en los registros C-HI-LO una posición hacia la derecha, y escribir un 0 en el bit C.

Se detallan a continuación los pasos del algoritmo para cada uno de los apartados del enunciado. Los ciclos se numeran desde el 0 (ciclo de inicialización) hasta el 5, ya que son 5 los bits empleados para los operandos. El resultado, de 10 bits, queda almacenado en los registros HI y LO al finalizar el último paso del algoritmo.

1. Multiplicando: $12_{10} = 01100_2$; Multiplicador: $5_{10} = 00101_2$.

Ciclo	Acción	C-HI-LO
0	Inicialización	0-00000-00101
1	$LO_0 = 1 \rightarrow$ Sumar M	0-01100-00101
1	Desplazar una posición hacia la derecha	0-00110-00010
2	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-00011-00001
3	$LO_0 = 1 \rightarrow$ Sumar M	0-01111-00001
3	Desplazar una posición hacia la derecha	0-00111-10000
4	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-00011-11000
5	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-00001-11100

Resultado: $0000111100_2 = 60_{10}$

2. Multiplicando: $18_{10} = 10010_2$; Multiplicador: $11_{10} = 01011_2$.

Ciclo	Acción	C-HI-LO
0	Inicialización	0-00000-01011
1	$LO_0 = 1 \rightarrow$ Sumar M	0-10010-01011
1	Desplazar una posición hacia la derecha	0-01001-00101
2	$LO_0 = 1 \rightarrow$ Sumar M	0-11011-00101
2	Desplazar una posición hacia la derecha	0-01101-10010
3	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-00110-11001
4	$LO_0 = 1 \rightarrow$ Sumar M	0-11000-11001
4	Desplazar una posición hacia la derecha	0-01100-01100
5	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-00110-00110

Resultado: $0011000110_2 = 198_{10}$

3. Multiplicando: $25_{10} = 11001_2$; Multiplicador: $19_{10} = 10011_2$.

Ciclo	Acción	C-HI-LO
0	Inicialización	0-00000-10011
1	$LO_0 = 1 \rightarrow$ Sumar M	0-11001-10011
1	Desplazar una posición hacia la derecha	0-01100-11001
2	$LO_0 = 1 \rightarrow$ Sumar M	1-00101-11001
2	Desplazar una posición hacia la derecha	0-10010-11100
3	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-01001-01110
4	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-00100-10111
5	$LO_0 = 1 \rightarrow$ Sumar M	0-11101-10111
5	Desplazar una posición hacia la derecha	0-01110-11011

Resultado: $0111011011_2 = 475_{10}$

4. Multiplicando: $30_{10} = 11110_2$; Multiplicador: $22_{10} = 10110_2$.

Ciclo	Acción	C-HI-LO
0	Inicialización	0-00000-10110
1	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-00000-01011
2	$LO_0 = 1 \rightarrow$ Sumar M	0-11110-01011
2	Desplazar una posición hacia la derecha	0-01111-00101
3	$LO_0 = 1 \rightarrow$ Sumar M	1-01101-00101
3	Desplazar una posición hacia la derecha	0-10110-10010
4	$LO_0 = 0 \rightarrow$ Sólo desplazar	0-01011-01001
5	$LO_0 = 1 \rightarrow$ Sumar M	1-01001-01001
5	Desplazar una posición hacia la derecha	0-10100-10100

Resultado: $1010010100_2 = 660_{10}$

■

PROBLEMA 16 Obtenga la codificación de Booth de los cuatro números binarios seguientes:

1. `0011 1011 0100 0011`
2. `1100 0100 1011 1100`
3. `0101 0110 1010 1011`
4. `1101 0101 1011 1100`

SOLUCIÓN En todos los casos se ha de asumir la existencia de un bit auxiliar, cuyo valor es cero y que está situado a la derecha del bit de menor peso. A partir de aquí, se ha de proceder aplicando la tabla de conversión de Booth tomando bits de dos en dos, desde el auxiliar hasta llegar al de mayor peso. En el primer paso se consideran el bit auxiliar y el de peso 2^0; en el segundo paso se consideran los bits de peso 2^0 y 2^1; en el tercero, el 2^1 y el 2^2, i así sucesivamente.

La tabla de conversión de Booth se adjunta a continuación:

q_i	q_{i-1}	Dígito Booth
0	0	0
0	1	1
1	0	-1
1	1	0

La solución a cada uno de los casos del enunciado es la siguiente:

1. `0100 -110-1 1-100 010-1`
2. `0-100 1-101 -1100 0-100`
3. `1-11-1 10-11 -11-11 -110-1`
4. `0-11-1 1-110 -1100 0-100`

■

PROBLEMA 17 Indique a qué cantidades (en decimal) corresponden los números siguientes expresados en código de Booth:

1. `0-101 00-10 10-10`

2. `-1001 0-110 0-100`
3. `0001 00-11 -110-1`
4. `1-100 010-1 01-10`

SOLUCIÓN La obtención de cada valor ha de hacerse considerando el número codificado según Booth como una cantidad representada en un sistema posicional donde el peso del dígito de posició i es 2^i, comenzando con un peso igual a $2^0 = 1$ para el dígito de menor pes. En otras palabras, cada dígito q_i aporta un valor $q_i \times 2^i$ a la cantitat representada.

El valor representado para cada uno de los casos del enunciado es el siguiente:

1. $(-1) \times 2^1 + 1 \times 2^3 + (-1) \times 2^5 + 1 \times 2^8 + (-1) \times 2^{10} = -794$
2. $(-1) \times 2^2 + 1 \times 2^5 + (-1) \times 2^6 + 1 \times 2^8 + (-1) \times 2^{11} = -1828$
3. $(-1) \times 2^0 + 1 \times 2^2 + (-1) \times 2^3 + 1 \times 2^4 + (-1) \times 2^5 + 1 \times 2^8 = 235$
4. $(-1) \times 2^1 + 1 \times 2^2 + (-1) \times 2^4 + 1 \times 2^6 + (-1) \times 2^{10} + 1 \times 2^{11} = 1074$

■

PROBLEMA 18 Considere el operador de multiplicación secuencial por Booth para números con signo y su algoritmo correspondiente. Suponga que el coste de las diferentes operaciones involucradas en el algoritmo es el siguiente:

- Inicializar registros y circuito de control: 2 ns.
- Inspeccionar q_i y q_{i-1}: 1 ns.
- Sumar: 9 ns.
- Restar: 10 ns.
- Desplazar `S-HI-LO-X`: 2 ns.
- Escribir registro `HI`: 2 ns.
- Evaluar el número de ciclo actual: 1 ns.

Suponga además que el paso de inicialización de los registros y del circuito de control se realiza en un ciclo de reloj aparte, antes de comenzar a procesar los bits del multiplicador.

1. ¿Cuál es el mínimo periodo aplicable a la señal de reloj?
2. Si se consideran operandos de 16 bits, ¿cuál es la productividad que puede proporcionar el multiplicador secuencial? Exprésela en MOPS.
3. ¿Cuál es la expresión general de la productividad del circuito expresada en MOPS en función del número de bits n?

SOLUCIÓN

1. Para determinar la duración mínima posible del ciclo de reloj es necesario considerar el retardo de cada una de les operaciones que han de realizarse en cada ciclo. En el caso que haya diversas alternativas –con este algoritmo ocurre, ya que es posible tener que hacer una suma, una resta o ninguna operación, según el valor de los bits q_i y q_{i-1}–, se considerará la operación con mayor retardo. El ciclo de reloj ha de adecuarse al caso que cueste más tiempo de resolver.

   En cada paso, el algoritmo ha de inspeccionar q_i y q_{i-1} (1 ns), posiblemente realizar una suma o una resta (en el peor caso una resta en 10 ns), desplazar los registros `S-HI-LO-X` un bit hacia la derecha (2 ns), escribir el nuevo valor en el registro `HI` (2 ns) y, finalmente, evaluar el número de ciclo actual (1 ns). La suma de los tiempos que cuestan estas operaciones es de 16 ns.

2. Siendo los operandos de 16 bits, serán necesarios un total de 17 ciclos para realizar la multiplicación: un ciclo de inicialización más 16 ciclos para procesar los 16 bits. La duración de cada ciclo se ha determinado en el apartado anterior (16 ns). El tiempo requerido para realizar una multiplicación será de 17 ciclos multiplicado por 16 ns, es decir, 272 ns. La productividad es la inversa del tempo que cuesta cada operación, es decir: $1/(272 \times 10^{-9}\ \text{s}) = 3676470{,}59$ operaciones por segundo $\simeq 3{,}68$ MOPS.

3. Teniendo en cuenta que el tiempo de ciclo está expresado en nanosegundos y que se ha de expresar el resultado en MOPS, la expresión de la productividad en función del número de bits es:

$$\frac{1}{16(n+1) \times 10^{-9} \times 10^{6}} = \frac{10^{3}}{16(n+1)}\ \text{MOPS}$$

El factor $n+1$ representa el número de ciclos necesarios para realizar una operación.

■

PROBLEMA 19 Considerando números de 4 bits, realice las multiplicaciones que se indican por medio del método de Booth, aplicando el operador secuencial y el algoritmo correspondientes:

1. $3 \times (-1)$
2. 5×3
3. $(-7) \times (-5)$
4. $(-4) \times 6$

SOLUCIÓN Para todos los casos, en primer lugar se ha de obtener la codificación binaria en complemento a dos de cada uno de los operandos. En el paso de inicialización, el registro M se carga con el valor del multiplicando, el multiplicador se almacena en el registro LO y el resto de bits (bit S, registro HI y bit auxiliar X) se inicializan a cero.

En cada paso del algoritmo se comienza por analizar el bit auxiliar (X) y el bit de menor peso del multiplicador (LO_0) para determinar la operación que se ha de hacer en este paso: sumar M a HI si $LO_0|X = 01$ o bien restar M a HI si $LO_0|X = 10$. Al final de cada paso, siempre se ha de desplazar el valor contenido en los registros S-HI-LO-X una posición hacia la derecha. Este desplazamiento hacia la derecha es de tipo aritmético, es decir, con copia del bit de signo S sobre sí mismo, con el fin de preservar el signo del valor almacenado en S-HI.

Se detallan a continuación los pasos del algoritmo para cada uno de los apartados del enunciado. Los ciclos se numeran desde el 0 (ciclo d'inicialización) hasta el 4, ya que son 4 los bits con que se representan los operandos. El resultado, de 8 bits, queda almacenado en los registros HI y LO al finalizar el último paso del algoritmo.

1. Multiplicando: $3_{10} = 0011_2$; Multiplicador: $-1_{10} = 1111_2$.

Ciclo	Acción	S-HI-LO-X
0	Inicialización	0-0000-1111-0
1	$LO_0\|X = 10 \rightarrow$ Restar M	1-1101-1111-0
1	Desplazar	1-1110-1111-1
2	$LO_0\|X = 11 \rightarrow$ Sólo desplazar	1-1111-0111-1
3	$LO_0\|X = 11 \rightarrow$ Sólo desplazar	1-1111-1011-1
4	$LO_0\|X = 11 \rightarrow$ Sólo desplazar	1-1111-1101-1

Resultado: $11111101_2 = -3_{10}$

2. Multiplicando: $5_{10} = 0101_2$; Multiplicador: $3_{10} = 0011_2$.

Ciclo	Acción	S-HI-LO-X
0	Inicialización	0-0000-0011-0
1	$LO_0 \vert X = 10 \rightarrow$ Restar M	1-1011-0011-0
1	Desplazar	1-1101-1001-1
2	$LO_0 \vert X = 11 \rightarrow$ Sólo desplazar	1-1110-1100-1
3	$LO_0 \vert X = 01 \rightarrow$ Sumar M	0-0011-1100-1
3	Desplazar	0-0001-1110-0
4	$LO_0 \vert X = 00 \rightarrow$ Sólo desplazar	0-0000-1111-0

Resultado: $00001111_2 = 15_{10}$

3. Multiplicand: $-7_{10} = 1001_2$; Multiplicador: $-5_{10} = 1011_2$.

Ciclo	Acción	S-HI-LO-X
0	Inicialización	0-0000-1011-0
1	$LO_0 \vert X = 10 \rightarrow$ Restar M	0-0111-1011-0
1	Desplazar	0-0011-1101-1
2	$LO_0 \vert X = 11 \rightarrow$ Sólo desplazar	0-0001-1110-1
3	$LO_0 \vert X = 01 \rightarrow$ Sumar M	1-1010-1110-1
3	Desplazar	1-1101-0111-0
4	$LO_0 \vert X = 10 \rightarrow$ Restar M	0-0100-0111-0
4	Desplazar	0-0010-0011-1

Resultado: $00100011_2 = 35_{10}$

4. Multiplicando: $-4_{10} = 1100_2$; Multiplicador: $6_{10} = 0110_2$.

Ciclo	Acción	S-HI-LO-X
0	Inicialización	0-0000-0110-0
1	$LO_0 \vert X = 00 \rightarrow$ Sólo desplazar	0-0000-0011-0
2	$LO_0 \vert X = 10 \rightarrow$ Restar M	0-0100-0011-0
2	Desplazar	0-0010-0001-1
3	$LO_0 \vert X = 11 \rightarrow$ Sólo desplazar	0-0001-0000-1
4	$LO_0 \vert X = 01 \rightarrow$ Sumar M	1-1101-0000-1
4	Desplazar	1-1110-1000-0

Resultado: $11101000_2 = -24_{10}$

■

PROBLEMA 20 Obtenga la recodificación por parejas de bits de los números siguientes, expresados en binario:

1. `0111 1011 0100 0011`
2. `1100 0100 1011 1100`
3. `0101 0110 1010 1011`
4. `1101 0101 1011 1100`

SOLUCIÓN Se ha de suponer la existencia de un bit auxiliar, cuyo valor es cero y que está situado a la derecha del bit de menor peso. A partir de aquí, se ha de proceder aplicando la tabla de recodificación por parejas, tomando los bits de tres en tres desde el bit auxiliar hasta llegar al de mayor peso. En el primer paso se consideran el bit auxiliar, el de peso 2^0 y el de peso 2^1 para obtener el dígito de recodificación por parejas de peso 2^0; en el segundo paso se consideran los bits de peso 2^1, 2^2 y 2^3 para obtener el dígito de peso 2^2; en el tercero el 2^3, el 2^4 y el 2^5 para obtener el dígito de peso 2^4 y así sucesivamente. Note que en la recodificación por parejas de bits se obtiene sólo un dígito por cada dos dígitos del número original.

La tabla de recodificación por parejas de bits se adjunta a continuación:

q_{i+1}	q_i	q_{i-1}	Recodificación
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

También es posible recodificar para parejas a partir de una recodificación previa de Booth. En este caso, simplemente se toman los dígitos de Booth de dos en dos desde el de menor peso (sin necesidad de considerar un dígito auxiliar) y se transforman cada dos dígitos de Booth en un dígito de parejas según el valor que aporten los dos dígitos de Booth a la cantidad representada por el número.

Las soluciones de cada uno de los casos del enunciado se presentan a continuación:

1. `0111 1011 0100 0011` $\equiv$ `2 0 -1 -1 1 0 1 -1`
2. `1100 0100 1011 1100` $\equiv$ `-1 0 1 1 -1 0 -1 0`
3. `0101 0110 1010 1011` $\equiv$ `1 1 2 -1 -1 -1 -1 -1`
4. `1101 0101 1011 1100` $\equiv$ `-1 1 1 2 -1 0 -1 0`

■

PROBLEMA 21 Indique a qué cantidades (en decimal) corresponden los números siguientes recodificados por parejas:

1. `2 -1 0 -2 1 -2`
2. `-1 1 2 -1 -2 1`
3. `-2 0 2 -1 1 0`
4. `0 1 1 -2 1 -1`

SOLUCIÓN Para obtener la cantidad representada por un número recodificado por parejas, ha de considerarse que este número está representado en un sistema posicional de pesos 1, 2, 4, 8, etc. Cada dígito del número representa su propio valor (0, 1, 2, -1 o -2) multiplicado por 2 elevado a la potencia que le corresponde según su posición en el número.

Los resultados para los números planteados en el enunciado son los siguientes:

1. $(-2) \times 2^0 + 1 \times 2^2 + (-2) \times 2^4 + (-1) \times 2^8 + 2 \times 2^{10} = 1762$
2. $1 \times 2^0 + (-2) \times 2^2 + (-1) \times 2^4 + 2 \times 2^6 + 1 \times 2^8 + (-1) \times 2^{10} = -663$
3. $1 \times 2^2 + (-1) \times 2^4 + 2 \times 2^6 + (-2) \times 2^{10} = -1932$

4. $(-1) \times 2^0 + 1 \times 2^2 + (-2) \times 2^4 + 1 \times 2^6 + 1 \times 2^8 = 291$

■

PROBLEMA 22 Obtenga los productos siguientes por el método de recodificación por parejas de bits del multiplicador, suponiendo que se utiliza el operador secuencial correspondiente. Considere operandos de 6 bits.

1. $30 \times (-6)$
2. 25×13
3. $(-17) \times (-19)$
4. $(-20) \times 22$

SOLUCIÓN Como son 6 los bits de los operandos, serán necesarios $6/2 = 3$ pasos del algoritmo, más el paso correspondiente a la inicialización de registros. En total quatre cicles de reloj. En cada ciclo i se analizarán los bits q_{i+1}, q_i y q_{i-1} del multiplicador, haciendo uso de un bit extra (X) que representa a q_{i-1}. Las operaciones a realizar dependen del valor de estos tres bits, y pueden ser sumar o restar M o 2M a HI o bien dejarlo intacto (en total, 5 posibilidades). Al final de cada ciclo se ha de realizar un desplazamiento aritmético de S-HI-LO-X de dos posiciones hacia la derecha.

Se presentan a continuación los resultados para cada uno de los casos del enunciado.

1. Multiplicando: $30_{10} = 011110_2$; Multiplicador: $-6_{10} = 111010_2$.

Cicle	Acció	S-HI-LO-X
0	Inicialización	0-000000-111010-0
1	$LO_{1,0}\|X = 100 \rightarrow$ Restar 2M	1-000100-111010-0
1	Desplazar dos posiciones a la derecha	1-110001-001110-1
2	$LO_{1,0}\|X = 101 \rightarrow$ Restar M	1-010011-001110-1
2	Desplazar dos posiciones a la derecha	1-110100-110011-1
3	$LO_{1,0}\|X = 111 \rightarrow$ Desplaçar	1-111101-001100-1

Resultado: $111101001100_2 = -180_{10}$

2. Multiplicando: $25_{10} = 011001_2$; Multiplicador: $13_{10} = 001101_2$.

Cicle	Acció	S-HI-LO-X
0	Inicialización	0-000000-001101-0
1	$LO_{1,0}\|X = 010 \rightarrow$ Sumar M	0-011001-001101-0
1	Desplazar dos posiciones a la derecha	0-000110-010011-0
2	$LO_{1,0}\|X = 110 \rightarrow$ Restar M	1-101101-010011-0
2	Desplazar dos posiciones a la derecha	1-111011-010100-1
3	$LO_{1,0}\|X = 001 \rightarrow$ Sumar M	0-010100-010100-1
3	Desplazar dos posiciones a la derecha	0-000101-000101-0

Resultado: $000101000101_2 = 325_{10}$

3. Multiplicando: $-17_{10} = 101111_2$; Multiplicador: $-19_{10} = 101101_2$.

Cicle	Acció	S-HI-LO-X
0	Inicialización	0-000000-101101-0
1	$LO_{1,0}\|X = 010 \rightarrow$ Sumar M	1-101111-101101-0
1	Desplazar dos posiciones a la derecha	1-111011-111011-0
2	$LO_{1,0}\|X = 110 \rightarrow$ Restar M	0-001100-111011-0
2	Desplazar dos posiciones a la derecha	0-000011-001110-1
3	$LO_{1,0}\|X = 101 \rightarrow$ Restar M	0-010100-001110-1
3	Desplazar dos posiciones a la derecha	0-000101-000011-1

Resultado: $000101000011_2 = 323_{10}$

4. Multiplicando: $-20_{10} = 101100_2$; Multiplicador: $22_{10} = 010110_2$.

Cicle	Acció	S-HI-LO-X
0	Inicialización	0-000000-010110-0
1	$LO_{1,0}\|X = 100 \rightarrow$ Restar 2M	0-101000-010110-0
1	Desplazar dos posiciones a la derecha	0-001010-000101-1
2	$LO_{1,0}\|X = 011 \rightarrow$ Sumar 2M	1-100010-000101-1
2	Desplazar dos posiciones a la derecha	1-111000-100001-0
3	$LO_{1,0}\|X = 010 \rightarrow$ Sumar M	1-100100-100001-0
3	Desplazar dos posiciones a la derecha	1-111001-001000-0

Resultado: $111001001000_2 = -440_{10}$

■

PROBLEMA 23 Considere el operador de multiplicación secuencial con recodificación por parejas de bits para números con signo y su algoritmo correspondiente. Suponga que el coste de las diferentes operaciones involucradas en el algoritmo es el siguiente:

- Inicializar registros y circuito de control: 2 ns.
- Inspeccionar q_{i+1}, q_i y q_{i-1}: 1 ns.
- Sumar: 9 ns.
- Restar: 10 ns.
- Desplazar `M`: 2 ns.
- Desplazar `S-HI-LO-X` 2 posiciones: 3 ns.
- Escribir registro `HI`: 2 ns.
- Evaluar el número de ciclo actual: 1 ns.

Suponga además que el paso de inicialización de los registros y del circuito de control se realiza en un ciclo de reloj aparte, antes de comenzar a procesar los bits del multiplicador.

1. ¿Cuál es el mínimo período aplicable a la señal de reloj?
2. Si se consideran operandos de 16 bits, ¿cuál es la productividad (en MOPS) que puede proporcionar el multiplicador secuencial?
3. ¿Cuál es la expresión general de la productividad del circuito expresada en MOPS en función del número de bits n?

SOLUCIÓN

1. De forma parecida a como se ha abordado el problema 18, para determinar la duración mínima posible del ciclo de reloj es necesario considerar el retardo de las operaciones que han de realizarse en cada ciclo. En el caso que haya diversas alternativas –con este algoritmo ocurre, ya que es posible tener que hacer una suma, una resta, o suma o resta con desplazamiento o ninguna operación, según el valor de los bits q_{i+1}, q_i y q_{i-1}–, se considerará la operación con mayor retardo, ya que el ciclo de reloj ha de adecuarse al caso que cueste mas tiempo resolver.

   En cada paso, el algoritmo ha de inspeccionar q_{i+1}, q_i i q_{i-1} (1 ns), posiblemente realizar un desplazamiento a la izquierda del multiplicando (para obtenir 2M, en 2 ns) y una suma o una resta (en el peor caso una resta en 10 ns), desplazar los registros `S-HI-LO-X` dos posiciones hacia la derecha (3 ns), escribir el nuevo valor en el registro `HI` (2 ns) y, finalmente, evaluar el número de ciclo actual (1 ns). La suma de lo que cuesta cada una de estas operaciones es de 19 ns.

2. Siendo los operandos de 16 bits, habrá un total de 9 ciclos para hacer la multiplicación: un ciclo de inicialización más $\frac{16}{2} = 8$ ciclos para procesar los 16 bits, ya que en cada ciclo se procesará un dígito que representa la recodificación de dos bits del multiplicador. La duración de cada ciclo se ha determinado en l'apartado anterior (19 ns). El tiempo requerido para hacer una multiplicación será de 9 ciclos multiplicado por 19 ns, es decir, 171 ns. La productividad es la inversa del tiempo que cuesta cada operación, es decir: $1/(171 \times 10^{-9}\ \text{s}) = 5847953{,}22$ operaciones por segundo $\simeq 5{,}85$ MOPS.

3. Teniendo en cuenta que el tiempo de ciclo está expresado en nanosegundos y que se ha de expresar el resultado en MOPS, la expresión de la productividad en función del número de bits és:

$$\frac{1}{19(\frac{n}{2}+1) \times 10^{-9} \times 10^{6}} = \frac{10^3}{19(\frac{n}{2}+1)}\ \text{MOPS}$$

   El factor $\frac{n}{2}+1$ representa el número de ciclos necesarios para hacer una operación de multiplicación completa.

■