Aritmètica d'enters

Índex

1 Operadors i operacions

PROBLEMA 1 En la Figura 1 teniu tres operadors de suma, que es descriuen de la manera següent:

1. El sumador A és combinacional i té un temps de retard de 10 ns.
2. L'operador B és seqüencial i se sincronitza amb un rellotge que oscil·la a 400 Mz. Per a fer una suma necessita 2 cicles.
3. L'operador C és també seqüencial i va governat per un rellotge que funciona a 150 MHz. En cada cicle pot fer dues sumes.

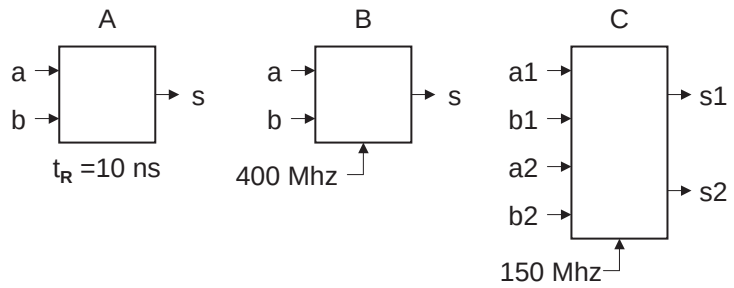

Figura 1: Tres operadors de suma. A és combinacional, B i C són seqüencials.

Calculeu quina és la productivitat màxima en MOPS de cada operador.

SOLUCIÓ

1. L'operador A és combinacional, amb un retard $t_R = 10$ ns. Si en la definició de productivitat considerem que la productivitat màxima s'obtindrà fent una operació cada t_R, escriurem:

$$\text{Productivitat màxima} = \frac{1 \text{ operació}}{10 \cdot 10^{-9} \text{ segons}} = 100 \cdot 10^6 \text{ operacions/segon} = 100 \text{ MOPS}$$

Dit d'altra manera, el nombre d'operacions per segon serà la inversa del retard del circuit.

2. L'operador B és seqüencial, i canvia d'estat a una freqüència de 400 MHz. Com que dos cicles de rellotge són el mínim temps per a fer una suma, podem adaptar la definició de la productivitat a les dades disponibles i escriure:

$$\text{Productivitat màxima} = \frac{400 \cdot 10^6 \text{ cicles/segon}}{2 \text{ cicles/operació}} = 200 \cdot 10^6 \text{ operacions/segon} = 200 \text{ MOPS}$$

   En aquest cas, la productivitat és la freqüència de rellotge dividida pel nombre de cicles per operació.

3. L'operador C pot fer com a màxim dues sumes per cicle. El càlcul és ara

$$\begin{aligned} \text{Productivitat màxima} &= 2 \text{ operacions/cicle} \times 150 \cdot 10^6 \text{ cicles/segon} \\ &= 300 \cdot 10^6 \text{ operacions/segon} = 300 \text{ MOPS} \end{aligned}$$

   En conseqüència, la productivitat obtinguda és el producte de la freqüència del rellotge pel nombre d'operacions per cicle.

■

PROBLEMA 2 Us han encomanat que dissenyeu un operador de suma per a un processador que treballa amb una freqüència de rellotge de 500 MHz. Per les característiques del processador, l'operador ha de fer una suma en cada cicle de rellotge, però només disposa de la meitat del període de rellotge per a fer-la. Quin serà el màxim retard possible de l'operador? Quina serà la productivitat màxima (en MOPS) de l'operador?

SOLUCIÓ El període de rellotge del processador es pot calcular per mitjà de l'expressió:

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

Així doncs, el temps de resposta de l'operador no podrà superar 1 ns.
La productivitat de l'operador serà el nombre d'operacions fetes per segon. En aquest cas,

$$\text{Productivitat} = 1 \text{ operació/cicle} \times 500 \cdot 10^6 \text{ cicles/segon} = 500 \cdot 10^6 \text{ operacions/segon} = 500 \text{ MOPS}$$

■

PROBLEMA 3 Considereu el codi següent escrit en llenguatge d'assemblador del MIPS R2000 que opera sobre un nombre de 32 bits contingut del registre `$t1` i deixa el resultat en el registre `$t0`.

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

1. Si els nombres estan representats en complement a dos, de quina operació es tracta? Justifiqueu-ne la resposta.

2. Quin seria el codi alternatiu si els nombres estigueren expressats en signe i magnitud?

SOLUCIÓ

1. El codi escriu en `$t0` el valor absolut del nombre contingut en `$t1`. En concret, primer copia `$t1` en `$t0`. Si `$t1` és positiu o igual a zero, aleshores salta a l'etiqueta i acaba el procés; en canvi, si `$t1` és negatiu, li canvia el signe (això és, el fa positiu) i el copia en `$t0`.

2. Si els nombres s'expressen en signe i magnitud n'hi haurà prou amb posar a zero el bit de signe del valor que hi ha al registre `$t1` i deixar el resultat en el registre `$t0`:

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

■

---

2 Suma, resta i comparació en el MIPS R2000

PROBLEMA 4 Escriviu en assemblador de MIPS R2000 el codi d'una funció que retorne el major dels seus dos paràmetres enters. Heu de seguir el conveni de programació usual: els paràmetres es troben en `$a0` i `$a1` i la funció ha de tornar el seu resultat en `$v0`. N'heu de fer dues versions:

1. Una funció `maxu` que opere amb valors sense signe, és a dir, que interprete els paràmetres en CBN.
2. Una funció `max` que opere amb valors amb signe, és a dir, que interprete els paràmetres en Ca2.

SOLUCIÓ Aquest exercici demana comparar els arguments de la funció atenent els tipus de dades. Les dues solucions són quasi idèntiques, només les instruccions de comparació són diferents perquè depenen del tipus de dades. Noteu que les solucions tornen `$a1` quan ambdós arguments són iguals.

1. Amb valors sense signe, farem servir `sltu` per a comparar els arguments:

```
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 es tracta de valors amb signe, farem servir la instrucció `slt` per a la comparació:

```
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 Considereu que esteu treballant en assemblador del MIPS R2000 i que teniu la necessitat de fer sumes i restes de nombres sense signe (en CBN) i amb signe (en Ca2), però que voleu detectar els desbordaments produïts en cada operació (sense provocar mai cap excepció) i, si és el cas, saltar a l'etiqueta `TractarOV` on està la part del codi que els tracta.

Escriviu en assemblador el codi que implemente les sentències d'alt nivell següents, on els símbols fan referència a variables de 32 bits. Les variables *var_a*, *var_b* i *var_c* no tenen signe i *var_d*, *var_e* i *var_f* sí en tenen.

1. $var\_a = var\_d$
2. $var\_d = var\_a$
3. $var\_a = var\_b + var\_c$
4. $var\_a = var\_b - var\_c$
5. $var\_d = var\_e + var\_f$
6. $var\_d = var\_e - var\_f$

SOLUCIÓ En les funcions que hem d'escriure no poden aparéixer les instruccions `add` i `sub` perquè en cas de desbordament aritmètic amb signe generen l'excepció OVF que provoca l'execució de la rutina de control d'excepcions del sistema. En el seu lloc, i per a ambdós tipus de dades, haurem d'utilitzar les instruccions `addu` i `subu`, que utilitzen els mateixos operadors però que no produeixen cap excepció.

1. Per a fer $var\_a = var\_d$ no cal cap operació de suma o de resta, només es tracta d'una assignació. Tanmateix, pot produir-se un desbordament si la variable amb signe var_d conté un valor negatiu. Així que el codi adient és el següent:

```
lw $t0,var_d
bltz $t0,Tractar_OV
sw $t0,var_a
```

   El codi anterior és equivalent a aquest altre:

```
lw $t0,var_d
slt $t1,$t0,$zero
bne $t1,$zero,Tractar_OV
sw $t0,var_a
```

   Una tercera alternativa podria ser l'anàlisi del bit de signe mitjançant operacions lògiques amb una màscara, com fa el codi següent:

```
lw $t0,var_d
li $t1,0x80000000
and $t1,$t1,$t0
bne $t1,$zero,Tractar_OV
sw $t0,var_a
```

2. L'operació $var\_d = var\_a$ pot produir desbordament si la variable sense signe var_a conté un valor major que el màxim positiu $7FFFFFFF_{16}$ codificable dins de la variable amb signe var_d. El codi següent comprova aquesta possibilitat i verifica que $var\_d < 80000000_{16}$:

```
lw $t0,var_a
li $t1,0x80000000
sltu $t2,$t0,$t1
beq $t2,$zero,Tractar_OV
sw $t0,var_d
```

   Podeu observar que les solucions mostrades en l'apartat anterior són totalment aplicables a aquest cas.

3. El desbordament propi d'una suma sense signe com $var\_a = var\_b + var\_c$ produeix un resultat que sempre és menor que qualsevol dels sumands. És a dir, que el valor $s = var\_b + var\_c$ calculat pel processador serà correcte si (i només si) $s \geq var\_b$. Podem escriure una primera solució que detecte desbordament comprovant la condició contrària $s < var\_b$ amb la instrucció `sltu`, que compara valors sense signe. Si falla la condició, el codi assigna $var\_a = s$, on s és el registre `$t2`.

```
lw $t0,var_b
lw $t1,var_c
addu $t2,$t0,$t1
sltu $t3,$t2,$t0
bne $t3,$zero,Tractar_OV
sw $t2,var_a
```

   Igualment, la suma serà correcta si (i només si) $s \geq var\_c$. Tenim, doncs, la solució anternativa següent:

```
lw $t0,var_b
lw $t1,var_c
addu $t2,$t0,$t1
sltu $t3,$t2,$t1
bne $t3,$zero,Tractar_OV
sw $t2,var_a
```

4. En el cas de la resta $var\_a = var\_b - var\_c$, sabrem que l'operació $r = var\_b - var\_c$ feta pel processador és correcta si (i només si) $r \leq var\_b$. Igualment, la resta és correcta si (i només si) $var\_b \geq var\_c$. Noteu que aquestes condicions són equivalents a les vistes en l'apartat anterior per a la suma si tenim en compte que $r = var\_b - var\_c \Rightarrow var\_b = r + var\_c$.

   En conseqüència, tenim dues solucions alternatives. La primera, comprova que $r \leq var\_b$ abans d'assignar el valor calculat a var_a:

```
lw $t0,var_b
lw $t1,var_c
subu $t2,$t0,$t1
sltu $t3,$t0,$t2
bne $t3,$zero,Tractar_OV
sw $t2,var_a
```

   La segona verifica que $var\_b \geq var\_c$ (o que $var\_c < var\_b$) abans de donar per bo el resultat calculat.

```
lw $t0,var_b
lw $t1,var_c
subu $t2,$t0,$t1
sltu $t3,$t0,$t1
bne $t3,$zero,Tractar_OV
sw $t2,var_a
```

5. El cas de la suma amb signe $var\_d = var\_e + var\_f$ és més complicat. Quan el processador calcula $s = var\_e + var\_f$, sabrem que l'operació ha produït un desbordament si els signes dels operands són iguals entre ells i, alhora, distints del signe del resultat.

   Amb aquesta idea, podem contruir una primera solució. Observeu el codi següent que, tot seguint la instrucció de suma `addu`, compara el signe dels operands. El procediment de comparació està basat en la instrucció `xor`, que en aquest context no té cap significat aritmètic, sinó que fa una comparació bit a bit entre els seus operands. Allò que importa és el bit de signe del resultat, que serà 0 si els signes dels operands són iguals i 1 en cas contrari. Amb una instrucció `and` i la màscara adient, es pot aïllar el bit clau: si és igual a 1 el codi pot donar el resultat per correcte, però en cas contrari caldrà seguir l'anàlisi perquè ens trobem en el cas d'operands amb el mateix signe: el codi ha de comparar els signes del resultat i d'un qualsevol dels dos operands mitjançant la instrucció `xor`.

```
        lw $t0,var_e
        lw $t1,var_f
        li $t4,0x80000000  # la màscara
        # primer cal sumar
        addu $t2,$t0,$t1
        # són iguals els signes dels operands?
        xor $t3,$t0,$t1
        and $t3,$t3,$t4
        bne $t3,$zero,OK # si distints, cap problema
        # atenció: coincideix el signe del resultat?
        xor $t3,$t0,$t2  # també podria ser xor $t3,$t1,$t2
        and $t3,$t3,$t4
        bne $t3,$zero,Tractar_OV
OK:     sw $t2,var_d
```

   Aquest codi es pot simplificar amb un truc de programació en assemblador. Per a bifurcar en funció del bit més significatiu d'un registre tenim prou amb les instruccions `bltz` i `bgez`, encara que el seu operand no tinga un significat numèric normal.

```
lw $t0,var_e
lw $t1,var_f
addu $t2,$t0,$t1
xor $t3,$t0,$t1
```

```
        bltz $t3,OK
        xor $t3,$t0,$t2   # també podria ser xor $t3,$t1,$t2
        bltz $t3,Tractar_OV
OK:     sw $t2,var_d
```

6. Per a tractar la resta de variables amb signe $var\_d = var\_e - var\_f$ cal comprovar que el resultat calculat $r = var\_e - var\_f$ és correcte adaptant els criteris de la suma al nostre cas. El resultat r ha de satisfer que $var\_e = r + var\_f$: si r i var_f tenen el mateix signe, aquest ha de coincidir amb el signe de var_e.

```
        lw $t0,var_e
        lw $t1,var_f
        subu $t2,$t0,$t1
        xor $t3,$t2,$t1
        bltz $t3,OK
        xor $t3,$t1,$t0
        bltz $t3,Tractar_OV
OK:     sw $t2,var_d
```

■

PROBLEMA 6 En tots els sistemes de representació binària amb n bits podeu trobar un màxim valor representable M i un mínim m. En un sistema de representació donat es defineix la suma amb saturació ss a l'operació definida així:

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

És a dir, la suma amb saturació coincideix amb la suma convencional mentre el resultat es trobe dins del rang $[m \ldots M]$; en cas de desbordament el resultat és el mínim o el màxim representable (el que estiga més aprop del resultat no saturat). De la mateixa manera s'hi pot definir la substracció amb saturació. La major part dels processadors actuals tenen instruccions que implementen aquest tipus d'aritmètica imprescindible per als gràfics, però el MIPS R2000 és massa antic per a això.

Heu de fer, en llenguatge d'assemblador, una biblioteca de funcions del R2000 amb les operacions següents:

1. `ss8u`: Suma amb saturació de nombres de 8 bits sense signe.
2. `ss8`: Suma amb saturació de nombres de 8 bits amb signe.
3. `ss32u`: Suma amb saturació de nombres de 32 bits sense signe.
4. `ss32`: Suma amb saturació de nombres de 32 bits amb signe.

Les funcions seguiran el conveni de pas de paràmetres pels registres `$a0` i `$a1` i tornaran el resultat en `$v0`. En els dos primers casos, les funcions podran suposar que els paràmetres rebuts estan ben formats, és a dir, que es troben dins del rang de representació de 8 bits corresponent: sense signe, els 24 bits més significatius dels arguments són zero; amb signe, els 25 bits més significatius són tots iguals i expressen el signe. Vegeu a la Figura 2 els formats utilitzats.

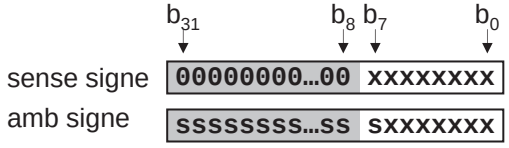

Figura 2: Formats de 8 bits. Els bits expressats com a `s` són tots iguals i representen el signe.

SOLUCIÓ

1. ss8u: Suma amb saturació de 8 bits sense signe.

   Per a fer la suma només tenim les instruccions convencionals add i addu. Podem triar qualsevol de les dues, perquè no hi pot donar-se una situació de desbordament detectable per addu. Una vegada feta la suma $x + y$, la funció haurà de comprovar que el resultat no desborda, és a dir, que $x + y$ es troba dins del rang $0 \ldots 255$. Si els valors no tenen signe, només poden donar-se casos de desbordaments del tipus $255 < (x + y)$. Vegeu en la Figura 3 aquest cas de desbordament.

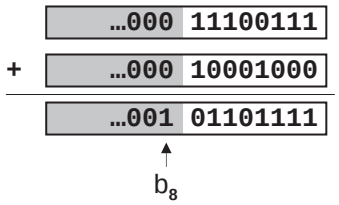

Figura 3: Cas de desbordament en l'aritmètica de 8 bits sense signe.

   Per a detectar el desbordament, la funció pot comparar el resultat amb 255 (usant la instrucció sltu perquè els operands no tenen signe). Si la suma supera aquest valor, cal substituir el resultat per 255; en cas contrari el resultat ja és correcte.

```
ss8u:     addu $v0,$a0,$a1
          sltiu $t0,$v0,256
          bne $t0,$zero,fi_ss8u
          li $v0,255
fi_ss8u:  jr $ra
```

   La detecció de la saturació es pot fer també mitjançant l'anàlisi del bit b_8 del resultat amb instruccions lògiques. Si $b_8 = 1$, s'ha produït la saturació.

```
ss8u:     addu $v0,$a0,$a1
          andi $t0,$v0,0x100
          beq $t0,$zero,fi_ss8u
          li $v0,255
fi_ss8u:  jr $ra
```

2. ss8: Suma amb saturació de 8 bits amb signe.

   L'aritmètica amb signe pot desbordar en ambdós sentits: si x i y són tots dos positius el resultat de la suma pot superar el màxim i entrar en el cas de $M < (x + y)$; si x i y són ambdós negatius, el resultat pot ser inferior al mínim i resultar $(x + y) < m$.

   Per a implementar ss8, cal tractar cada cas de desbordament per separat. En aquest cas, tenim $m = -128$ i $M = +127$. Vegeu exemples dels dos casos de desbordament en la Figura 4.

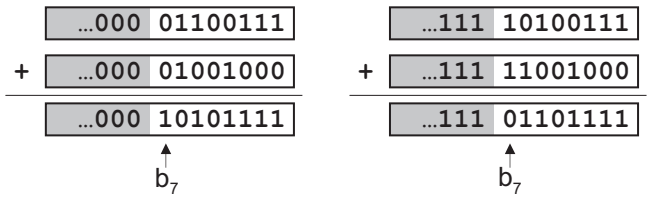

Figura 4: Dos casos de saturació amb signe.

   La detecció del desbordament pot fer-se amb dues comparacions, una per cada extrem del rang, com en aquesta solució:

```
ss8:      addu $v0,$a0,$a1
perdalt:  slti $t0,$v0,128
```

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

També s'hi pot detectar el desbordament per l'anàlisi dels bits b_7 de signe. La situació es dóna quan el signe dels dos sumands és el mateix i, alhora, és distint del signe del resultat. En aquest cas, cal interpretar que si el signe del resultat calculat és positiu el desbordament s'ha produït pel límit inferior del rang; en cas contrari, el desbordament s'ha produït pel límit superior. La solució que es mostra tot seguit segueix aquestes fases:

```
ss8:      addu $v0,$a0,$a1
          # analitzem els signes de $a0 i $a1
          andi $t0,$a0,0x80
          andi $t1,$a1,0x80
          bne $t0,$t1,fi_ss8
          # alerta: signes de $a0 i $a1 iguals
          # cal mirar el signe de $v0
          andi $t1,$v0,0x80
          beq $t0,$t1,fi_ss8
          # confirmat: signe de $v0 distint
          # cal saturar en funció del signe de $v0:
          beq $t1,$zero,perbaix
perdalt:  li $v0,127
          j fi_ss8
perbaix:  li $v0,-128
fi_ss8:   jr $ra
```

3. ss32u: Suma amb saturació de 32 bits sense signe.

   En realitat, el problema de la suma de 32 bits amb saturació amb variables sense signe el tenim quasi resolt en l'apartat 3 del problema 5. La diferència és que ara hem de substituir el resultat desbordat de la suma pel valor límit $M = \text{FFFFFFFF}_{16}$. Adaptant una de les solucions que hi escrivírem, tenim:

```
ss32u:    addu $v0,$a0,$a1
          sltu $t0,$v0,$a0
          beq $t0,$zero,fi_ss32u
          li $v0,0xffffffff
fi_ss32u: jr $ra
```

4. ss32: Suma amb saturació de 32 bits amb signe.

   Per a la suma amb saturació de 32 bits amb signe, podem adaptar la solució de l'apartat 5 del problema 5. La solució que mostrem més endavant fa la mateixa anàlisi dels signes buscant el desbordament. En detectar el desbordament, caldrà tractar per separat els dos casos possibles: que el resultat siga $(x + y) > M$ o que siga $(x + y) < m$. Per a distingir-los, podem analitzar el bit de signe del resultat calculat, tot considerant que el desbordament sempre inverteix aquest bit. Si el resultat hi apareix com a positiu, haurem d'interpretar que el càlcul ha desbordat per l'extrem inferior, és a dir, que $(x + y) < m$; en cas contrari sabrem que ha desbordat per l'extrem superior i, aleshores, que $(x + y) > M$. Després caldrà ajustar el resultat a m en el primer cas i a M en el segon.

```
ss32:     addu $v0,$a0,$a1
          # detecció del desbordament
          xor $t0,$a0,$a1
          bltz $t0,fi_ss32
          xor $t0,$a0,$v0
          bgez $t0,fi_ss32
```

```
          # per a on s'ha desbordat?
          bgez $v0,perbaix
          # ajust
perdalt:  li $v0,0x7fffffff
          j fi_ss32
perbaix:  li $v0,0x80000000
fi_ss32:  jr $ra
```

■

PROBLEMA 7 Escriviu en assemblador un programa que sume dues variables *var_b* i *var_c* i assigne el resultat a *var_a*. Totes tres variables són del tipus *long* de Java (enter de 64 bits). Com que el MIPS R2000 no suporta aquest tipus de dades, fixarem un conveni d'ús abans de començar:

- Cada variable s'ubicarà en dues paraules consecutives de memòria, que tindran adreces A i $A+4$; la part menys significativa de la variable s'emmagatzemarà en A i la part més significativa en $A+4$.
- El conveni de comunicació amb el programa cridant és el següent: el primer paràmetre es troba en `$a1`‖`$a0`; és a dir, que els 32 bits més significatius es troben en `$a1` i els menys significatius en `$a0`. El segon argument es troba en `$a3`‖`$a2`. La funció ha de tornar el seu resultat en `$v1`‖`$v0`.

Aplicant el conveni descrit, seguiu les passes següents per a arribar a la solució completa:

1. Escriviu en assemblador una funció `sumDPu` que calcule la suma de dos paràmetres de 64 bits que no provoque excepció en cas de desbordament.
2. Escriviu en assemblador la secció de dades del programa, on han d'estar ubicades les tres variables *var_a*, *var_b* i *var_c*. Els valors inicials han de ser:

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

3. Completeu el programa escrivint el codi executable principal que llegisca de la memòria les variables *var_b* i *var_c*, cride `sumDPu` per sumar-les i escriga en *var_a* el resultat.
4. Què canviaríeu de tot el programa que heu escrit en els apartats anteriors si les variables tingueren signe i volguéreu que els desbordaments generaren una excepció OVF?

SOLUCIÓ

1. Disseny de la funció `sumDPu`:

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

2. Especificació de les variables. Com l'assemblador del MIPS R2000 ubica els objectes descrits en l'arxiu font per ordre creixent d'adreces, en l'àrea de variables, tot seguint l'etiqueta, caldrà especificar primer la paraula menys significativa i després la més significativa. Vegeu com es descriuen les variables d'aquest problema:

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

3. En el cos del programa principal cal carregar els registres que fan el pas de paràmetres a la funció. Noteu com, per a cadascun d'ells, la paraula menys significativa s'obté mitjançant l'etiqueta, mentre que la paraula més significativa és accessible amb un desplaçament de 4. D'igual manera es tracta el resultat de la funció.

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

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

# crida a sumDP
jal sumDPu

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

4. Per a un tractament de l'aritmètica amb signe equivalent al de la instrucció `add`, només caldria modificar la funció. Vegeu el codi de la funció resultant `sumDP`: hi cal substituir les dues instruccions de suma que calculen la part més significativa del resultat. El desbordament pot donar-se en dos moments: quan se suma la contribució del transport al primer operand i quan se suma el segon operand. En ambdós casos, la instrucció adient és `add` en comptes de `addu`.

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

■

PROBLEMA 8 Escriviu en assemblador de MIPS R2000 el codi d'una funció `maxDP` que retorne el major dels seus dos arguments tipus *long* de Java (enter de 64 bits). Seguiu el conveni de programació definit en el problema 7.

SOLUCIÓ Per al disseny de `maxDP` podem tractar els arguments començant per la seua part més significativa amb la instrucció `sltu`. La comparació es pot resoldre de seguida si les parts més significatives d'ambdós operands no són iguals. En el cas que siguen iguals, caldrà comparar les parts menys significatives. Totes dues comparacions es duen a terme amb la instrucció `sltu`.

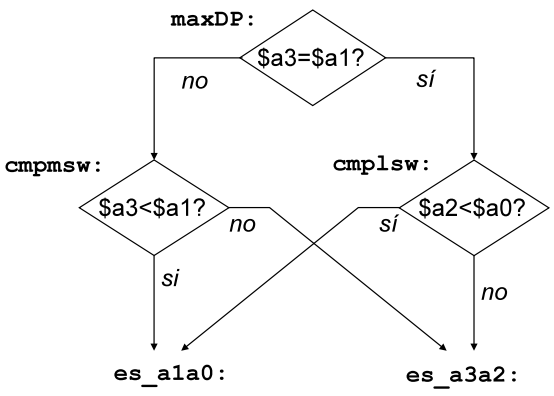

Figura 5: Arbre de decisions de `maxDP`.

La Figura 5 il·lustra l'arbre de decisions que guia el codi de la solució que es mostra tot seguit. En l'etiqueta es_a3a2 està codificat el tractament aplicable si $a3||$a2>$a1||$a0 i en es_a3a2 el cas contrari. Noteu que si ambdós arguments són iguals, la funció retorna $a3||$a2.

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

■

3 Disseny d'operadors de suma i resta

PROBLEMA 9 Les Figures 6 i 7 mostren dues implementacions del sumador complet. El circuit de la Figura 6 es dedueix directament de les funcions lògiques i el de la Figura 7 combina dos semisumadors amb una porta *or*.

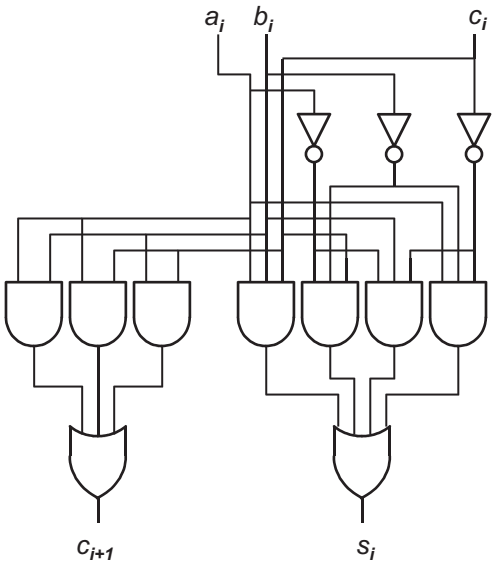

Figura 6: Implementació del sumador d'un bit a partir de les funcions lògiques.

Considereu els retards de les portes llistades a la taula següent:

Porta	Retard (ns)
not	0.5
and i or de 2 entrades	1
exor de 2 entrades	2
and i or de 3 entrades	1.2
or de 4 entrades	1.5

Amb aquestes dades, calculeu els retards de cadascuna de les eixides dels circuits següents:

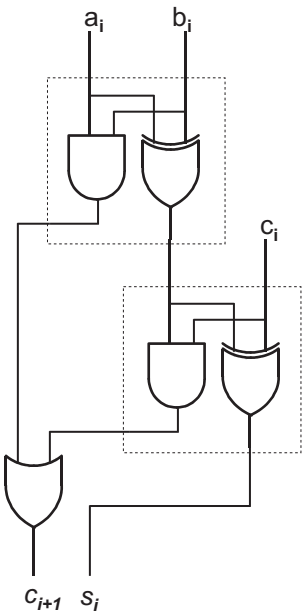

Figura 7: Implementació del sumador d'un bit mitjançant semisumadors.

1. El sumador d'un bit mostrat en la Figura 6.
2. El sumador d'un bit mostrat en la Figura 7.
3. Un sumador de 4 bits construït amb sumadors com els de la Figura 6 connectats en sèrie.
4. Sumador de 4 bits construït amb sumadors com els de la Figura 7 connectats en sèrie.

Solució

1. Farem l'anàlisi del sumador complet implementat a partir de les funcions lògiques mostrat en la Figura 6. Aquest circuit té dues eixides: el bit de suma s_i i l'eixida de transport c_{i+1}.

   - **bit de suma s_i:** El nivell d'aquesta eixida és 3. El retard és de $t_S = t_{Not} + t_{And(3)} + t_{Or(4)} = 0.5 + 1.2 + 1.5 = 3.2$ ns
   - **bit de transport c_{i+1}:** Aquesta eixida és de nivell 2, i el retard $t_C = t_{And(2)} + t_{Or(3)} = 1 + 1.2 = 2.2$ ns

   Noteu que, per simetria, aquests retards són iguals per a totes les entrades. És a dir, un canvi en qualsevol entrada a_i, b_i o c_i tindrà efecte en les eixides després dels retards calculats.

2. Quant a la implementació mitjançant semisumadors, els retards són:

   - **bit de suma s_i:** El nivell d'aquesta eixida és 2. En ambdós nivells, formats per semisumadors (vegeu la Figura 7), la porta *exor* determina el retard. El retard total és de $t_S = 2 \times t_{Exor} = 4$ ns.
   - **bit de transport c_{i+1}:** Aquesta eixida és de nivell 3. En el primer nivell està el semisumador superior de la Figura 7 amb un retard de t_{Exor}, en el segon nivell està el semisumador inferior amb un retard de $t_{And(2)}$ i en el tercer hi ha la porta *or*. Sumant-ho tot, resulta $t_C = t_{Exor} + t_{And(2)} + t_{Or(2)} = 4$ ns.

   En aquesta implementació ha desaparegut la simetria entre les entrades. Un canvi en una entrada a_i o b_i tindrà efecte en les eixides s_i i c_{i+1} després dels retards t_C i t_S calculats més amunt; però un canvi en l'entrada c_i, tot mantenint les altres entrades estables, afectarà les eixides amb uns retards relatius menors. El retard relatiu de l'eixida s_i respecte de l'entrada c_i l'expressarem com $t_{C \to S} = t_{Exor} = 2$ ns; el retard relatiu de c_{i+1} és $t_{C \to C} = t_{And} + t_{Or} = 2$ ns. Aquests retards relatius són vàlids $t_{Exor} = 2$ ns o més després d'haver-se estabilitzat les entrades a_i o b_i.

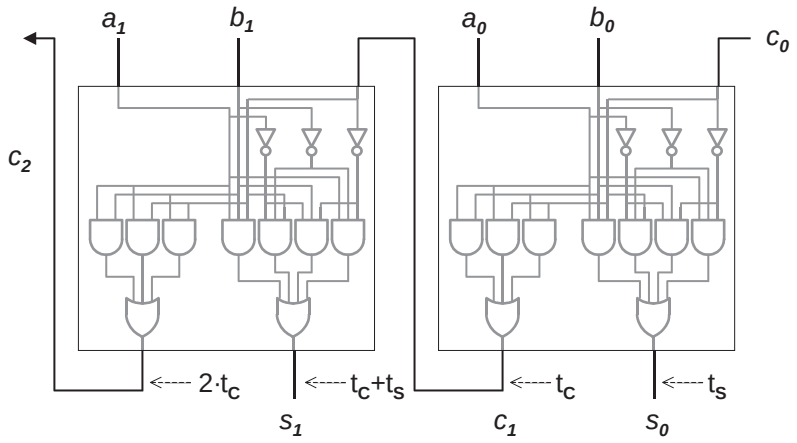

Figura 8: Les dues etapes menys significatives del sumador sèrie de 4 bits.

3. Per a l'anàlisi temporal del sumador sèrie de 4 bits construït amb sumadors implementats a partir de les funcions lògiques només cal considerar els retards t_S i t_C calculats en l'apartat 1, com es mostra en la Figura 8.

   En general, el retard dels bits de transport c_i és de $t_C \cdot i$ i el retard de les eixides de suma s_i és $t_C \cdot i + t_S$. Aplicant els resultats de l'apartat 1, tenim els retards de les eixides, junt amb els retards dels transports intermedis, en la taula següent:

Senyal	Retard (ns)
s_0	3.2
s_1	5.4
s_2	7.6
s_3	9.8

Senyal	Retard (ns)
c_1	2.2
c_2	4.4
c_3	6.6
c_4	8.8

4. Vegeu en la Figura 9 l'anàlisi del sumador sèrie de 4 bits construït amb semisumadors. Els retards de l'eixida s_0 i del transport c_1 són els valors $t_S = 4$ ns i $t_C = 4$ ns calculats en l'apartat 2. Per a calcular els retards de l'eixida s_1 i del transport c_2 a partir del moment en què c_1 pren el seu valor, hem d'entendre que les entrades a_1 i b_1 porten més de 2 ns estables i que per això podem aplicar-hi els retards $t_{C\to C}$ i $t_{C\to S}$ relatius al transport. Generalitzant, podem escriure el retard dels bits de transport c_i $(i > 0)$ com $t_C + t_{C\to C} \cdot (i-1)$ i el retard de les eixides de suma s_i $(i > 0)$ com $t_C + t_{C\to C} \cdot (i-1) + t_{C\to S}$.

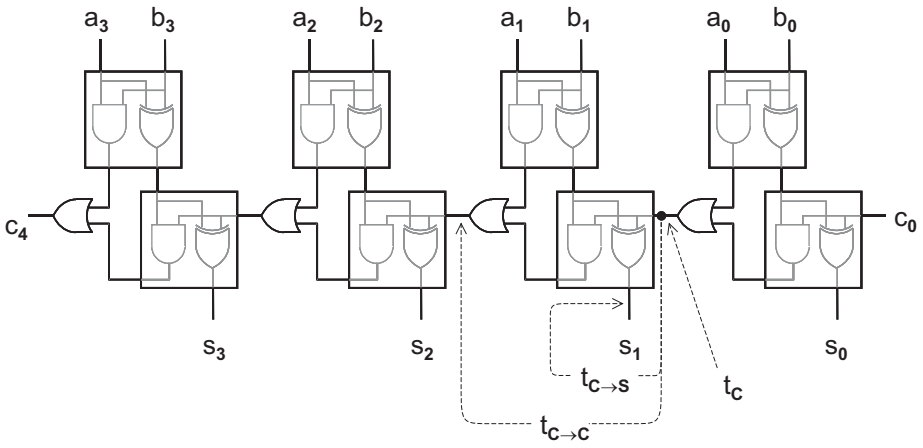

Figura 9: Sumador sèrie de 4 bits fet amb semisumadors.

Donant valors numèrics, obtenim els retards de les eixides i dels transports:

Senyal	Retard (ns)
s_0	4
s_1	6
s_2	8
s_3	10

Senyal	Retard (ns)
c_1	4
c_2	6
c_3	8
c_4	10

■

PROBLEMA 10 Hi ha una tècnica de disseny de sumadors ràpids anomenada *Carry Select Adder*, que adaptarem als nostres dissenys. Es tracta d'estructurar l'operador de n bits en seccions de m bits que treballen en paral·lel. En cada secció hi ha dos sumadors de m bits que calculen la suma dels bits implicats per duplicat: un operador assumeix que el transport d'entrada a la secció és 0 i l'altre que és 1. Quan s'ha determinat el valor real del transport d'entrada a la secció, un multiplexor selecciona el resultat correcte. Vegeu a la Figura 10 un cas amb $n = 32$ bits i $m = 16$ bits.

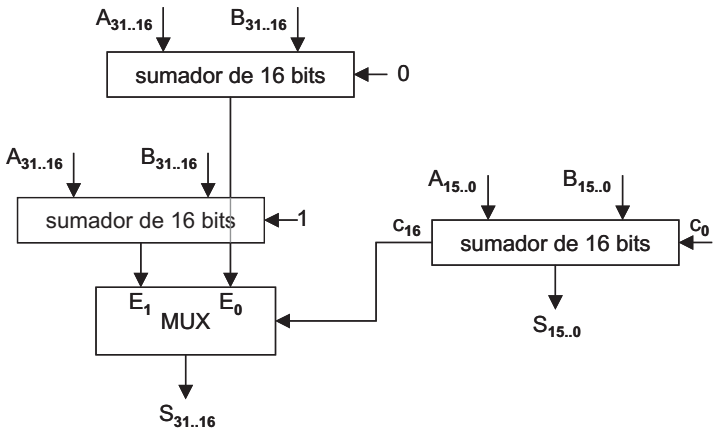

Figura 10: Aplicació de la tècnica *Carry Select Adder* a un sumador de 32 bits.

Calculeu el retard i la productivitat del circuit de la Figura 10. Considereu que les portes tenen un retard d'1 ns, que els sumadors estan implementats a partir de les funcions lògiques i que el multiplexor és un circuit de nivell 3. Amb aquesta disposició, utilitzeu sumadors CPA de 16 bits.

SOLUCIÓ Atés que l'eixida de transport d'un sumador complet té un retard de $t_C = 2$ ns, el retard de l'eixida c_{16} d'un sumador CPA de $n = 16$ bits serà $t_{op} = n \cdot t_C = 32$ ns; el retard de les eixides de suma $s_{15} \ldots s_0$, com vam veure al problema **??**, és de 33 ns.

En aquest cas de sumador de 32 bits, caldrà considerar el retard addicional de 3 ns a causa del multiplexor, que comptarem a partir de l'instant $t = 33$ ns en què se li apliquen totes les entrades. En total, el retard és de 36 ns, i la productivitat resultant serà de 27.8 MOPS.

■

---

4 Multiplicació amb el MIPS R2000

PROBLEMA 11 Heu de fer en llenguatge d'assemblador del MIPS R2000 una funció amb l'especificació següent:

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

La funció no pot fer ús de les instruccions de multiplicació del MIPS i només podrà utilitzar instruccions de suma, resta i desplaçament. No cal detectar desbordaments aritmètics.

SOLUCIÓ En general, per a multiplicar en binari per la potència n-èsima de 2, cal fer un desplaçament de n posicions cap a l'esquerra. Això és precisament el que fa la instrucció MIPS `sllv`:

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

■

PROBLEMA 12 Heu de fer en llenguatge d'assemblador una biblioteca de funcions del MIPS R2000 amb les especificacions següents:

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

Les rutines no poden fer ús de les instruccions de multiplicació del MIPS i només utilitzaran instruccions de suma, resta i desplaçament. No cal detectar desbordaments aritmètics.

SOLUCIÓ

1. `mul4`: El mètode més directe per a multiplicar per quatre és el desplaçament de l'operand cap a l'esquerra en dos posicions.

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

2. `mul5`: Ací cal multiplicar `$a0` per una constant que no és potència de dos. Tanmateix, podem descompondre el factor 5 en suma de dues potències del dos: $4 + 1$. Per la propietat distributiva del producte respecte de la suma, tenim que `$a0` $\times 5 =$ `$a0` $\times (4 + 1)$, com queda reflectit en el codi següent:

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

3. `mul10`: Per a multiplicar per 10, podem aprofitar la multiplicació per 5 feta en l'apartat 2 i multiplicar el resultat per 2 i escriure la funció següent:

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

   Si descomponem 10 en suma de potències de 2 tenim $10 = 8 + 2 = 2^3 + 2^1$. Tot distribuint el producte entre els sumands, podem escriure aquesta nova versió de la funció:

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

   Aquesta tècnica de multiplicació d'una variable per una constant amb desplaçaments i sumes només és còmoda quan la constant té pocs bits a 1. Noteu que per a una constant que conté n bits a 1 caldrà fer $n - 1$ sumes i n desplaçaments. (Podem estalviar un desplaçament si apareix 2^0 en la descomposició, com ens ha passat en l'apartat 2.)

4. `mul15`: Una primera solució, basada en la descomposició de 15 en suma de potències de 2: $15 = 2^3 + 2^2 + 2^1 + 2^0$ seria aquesta:

```
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
```

Noteu que hi fan falta 3 sumes i 3 desplaçaments, com correspon a la descomposició del 15 en quatre potències del 2.

■

PROBLEMA 13 En el problema 12 s'han dissenyat funcions que multipliquen valors sense signe sense detectar desbordaments. Com podrien aquestes funcions detectar el desbordament?

SOLUCIÓ Per a detectar desbordaments amb operands enters sense signe no podem adaptar la tècnica aplicada en el problema 5 (apartat 3). No és prou comparar el producte calculat amb el factor `$a0`. Per exemple, si la funció calcula $m = 4 \times$ `$a0`, quan `$a0` $= 60000000_{16}$ resultaria $m = 80000000_{16}$, i s'acompleix que $m >$ `$a0` encara que l'operació ha desbordat.

En un desplaçament cap a l'esquerra la verificació ha de fer-se comprovant que els bits eliminats són tots zeros (recordeu que estem treballant amb operands sense signe). En l'exemple de la multiplicació per 4, caldrà comprovar que els dos bits més significatius de `$a0` són zero, ja que `$a0` s'haurà de desplaçar 2 posicions a l'esquerra per a multiplicar-lo per 4:

```
# codi per a multiplicar per 4 (valors sense signe)
        sll $v0,$a0,2
        # afegit per a detectar desbordaments
        li $t0,0xC0000000
        and $t0,$t0,$a0
        bne $t0,$zero,Tractar_Desbordament
```

En productes més complicats (per exemple, en una multiplicació per 10) caldrà verificar tots els desplaçaments que es facen. També caldrà verificar les sumes i les restes amb comparacions com les de l'apartat 3 del problema 5.

Per últim, convé recordar que la comprovació de desbordament en la multiplicació no és necessària si es reserven $2n$ bits per al producte.

■

PROBLEMA 14 Considereu que esteu treballant amb un MIPS R2000 a una freqüència de rellotge de 100 MHz i que totes les instruccions d'enters s'executen en un cicle, excepte les instruccions generals de multiplicació `mult` i `multu` que en demanen 6 cicles.

1. Quin és el temps d'execució de les funcions que hem escrit en el problema 12?
2. Quin seria el temps d'execució de les funcions si s'hi hagueren utilitzat les instruccions generals de multiplicació?

Considereu que no hi cal detectar desbordament i que els resultats de tots els productes caben en un registre de 32 bits.

SOLUCIÓ

1. Les funcions que hem escrit en el problema 12 necessiten $10 \times n$ ns per a executar-se, on n és el nombre d'instruccions que contenen. Per exemple, la solució a l'apartat 4 (`mul15`), amb 7 instruccions, té un temps d'execució de 70 ns.
2. Prenguem com a exemple la multiplicació per 15 per mitjà de la instrucció `mul`. La implementació seria aquesta:

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

El temps d'execució seria ara de $60 + 3 \times 10 = 90$ ns. Qualsevol funció que continga menys de 9 instruccions, comptant entre sumes, restes, desplaçaments i el retorn final, serà més ràpida que aquesta última, com és el cas de totes les solucions del problema 12.

En les seues recomanacions per a optimització de codi, els dissenyadors dels processadors solen aconsellar que els compiladors tradueixquen les multiplicacions per constants petites conegudes en el moment de la compilació mitjançant les tècniques que hem practicat en el problema 12.

■

---

5 Operadors de multiplicació

PROBLEMA 15 Aplicant l'algorisme de multiplicació de nombres sense signe per mitjà d'un operador seqüencial, feu els productes següents en base 2 i utilitzant 5 bits:

1. 12×5
2. 18×11
3. 25×19
4. 30×22

SOLUCIÓ Per a tots els casos, en primer lloc s'ha d'obtenir la codificació binària de cada un dels operands. En el pas d'inicialització, el registre M es carrega amb el valor del multiplicand, el multiplicador s'emmagatzema en el registre LO i el bit C s'inicialitza a zero.

En cada pas de l'algorisme es comença per analitzar el bit de menor pes del multiplicador (LO_0) per a determinar si s'ha de sumar M a HI (si $LO_0 = 1$) o no (si $LO_0 = 0$). Al final de cada pas, sempre s'ha de desplaçar el valor contingut en els registres C-HI-LO una posició cap a la dreta, i escriure un 0 al bit C.

Es detallen a continuació els passos de l'algorisme per a cadascun dels apartats de l'enunciat. Els cicles es numeren des del 0 (cicle d'inicialització) fins al 5, ja que són 5 els bits emprats per als operands. El resultat, de 10 bits, queda emmagatzemat en els registres HI i LO en finalitzar l'últim pas de l'algorisme.

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

Cicle	Acció	C-HI-LO
0	Inicialització	0-00000-00101
1	$LO_0 = 1 \rightarrow$ Sumar M	0-01100-00101
1	Desplaçar una posició cap a la dreta	0-00110-00010
2	$LO_0 = 0 \rightarrow$ Només desplaçar	0-00011-00001
3	$LO_0 = 1 \rightarrow$ Sumar M	0-01111-00001
3	Desplaçar una posició cap a la dreta	0-00111-10000
4	$LO_0 = 0 \rightarrow$ Només desplaçar	0-00011-11000
5	$LO_0 = 0 \rightarrow$ Només desplaçar	0-00001-11100

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

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

Cicle	Acció	C-HI-LO
0	Inicialització	0-00000-01011
1	$LO_0 = 1 \rightarrow$ Sumar M	0-10010-01011
1	Desplaçar una posició cap a la dreta	0-01001-00101
2	$LO_0 = 1 \rightarrow$ Sumar M	0-11011-00101
2	Desplaçar una posició cap a la dreta	0-01101-10010
3	$LO_0 = 0 \rightarrow$ Només desplaçar	0-00110-11001
4	$LO_0 = 1 \rightarrow$ Sumar M	0-11000-11001
4	Desplaçar una posició cap a la dreta	0-01100-01100
5	$LO_0 = 0 \rightarrow$ Només desplaçar	0-00110-00110

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

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

Cicle	Acció	C-HI-LO
0	Inicialització	0-00000-10011
1	$LO_0 = 1 \rightarrow$ Sumar M	0-11001-10011
1	Desplaçar una posició cap a la dreta	0-01100-11001
2	$LO_0 = 1 \rightarrow$ Sumar M	1-00101-11001
2	Desplaçar una posició cap a la dreta	0-10010-11100
3	$LO_0 = 0 \rightarrow$ Només desplaçar	0-01001-01110
4	$LO_0 = 0 \rightarrow$ Només desplaçar	0-00100-10111
5	$LO_0 = 1 \rightarrow$ Sumar M	0-11101-10111
5	Desplaçar una posició cap a la dreta	0-01110-11011

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

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

Cicle	Acció	C-HI-LO
0	Inicialització	0-00000-10110
1	$LO_0 = 0 \rightarrow$ Només desplaçar	0-00000-01011
2	$LO_0 = 1 \rightarrow$ Sumar M	0-11110-01011
2	Desplaçar una posició cap a la dreta	0-01111-00101
3	$LO_0 = 1 \rightarrow$ Sumar M	1-01101-00101
3	Desplaçar una posició cap a la dreta	0-10110-10010
4	$LO_0 = 0 \rightarrow$ Només desplaçar	0-01011-01001
5	$LO_0 = 1 \rightarrow$ Sumar M	1-01001-01001
5	Desplaçar una posició cap a la dreta	0-10100-10100

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

■

PROBLEMA 16 Obteniu la codificació de Booth dels quatre nombres binaris següents:

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

SOLUCIÓ En tots els casos s'ha d'assumir l'existència d'un bit auxiliar, el valor del qual és zero i que està situat a la dreta del bit de menor pes. A partir d'ací, ha de procedir-se aplicant la taula de conversió de Booth prenent bits de dos en dos, des de l'auxiliar fins a arribar al de major pes. En el primer pas es consideren el bit auxiliar i el de pes 2^0; en el segon pas se consideren els bits de pes 2^0 i 2^1; en el tercer, el 2^1 i el 2^2, i així successivament.

La taula de conversió de Booth s'adjunta a continuació:

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

La solució a cadascun dels casos de l'enunciat és la següent:

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 Indiqueu a quines quantitats (en decimal) corresponen els nombres següents expressats en codi 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`

SOLUCIÓ L'obtenció de cada valor ha de fer-se considerant el nombre codificat segons Booth com una quantitat representada en un sistema posicional que el pes del dígit de posició i és 2^i, començant amb un pes igual a $2^0 = 1$ per al dígit de menor pes. En altres paraules, cada dígit q_i aporta un valor $q_i \times 2^i$ a la quantitat representada.

El valor representat per a cadascun dels casos de l'enunciat és el següent:

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 Considereu l'operador de multiplicació seqüencial per Booth per a nombres amb signe i el seu algorisme corresponent. Suposeu que el cost de les diferents operacions involucrades en l'algorisme és el següent:

- Inicialitzar registres i circuit de control: 2 ns.
- Inspeccionar q_i i q_{i-1}: 1 ns.
- Sumar: 9 ns.

- Restar 10 ns.
- Desplaçar `S-HI-LO-X`: 2 ns.
- Escriure registre `HI`: 2 ns.
- Avaluar el nombre de cicle actual: 1 ns.

Suposeu a més que el pas d'inicialització dels registres i del circuit de control es realitza en un cicle de rellotge a banda, abans de començar a processar els bits del multiplicador.

1. Quin és el mínim període aplicable al senyal de rellotge?
2. Si es consideren operands de 16 bits, quina és la productivitat que pot proporcionar el multiplicador seqüencial? Expresseu-la en MOPS.
3. Quina és l'expressió general de la productivitat del circuit expressada en MOPS en funció del nombre de bits n?

SOLUCIÓ

1. Per a determinar la duració mínima possible del cicle de rellotge és necessari considerar el retard de cada una de les operacions que han de realitzar-se en cada cicle. En el cas que hi haja diverses alternatives –amb aquest algorisme ocorre, ja que és possible haver de fer una suma, una resta o cap operació, segons el valor dels bits q_i i q_{i-1}–, es considerarà l'operació amb major retard. El cicle de rellotge ha d'adequar-se al cas que coste més temps resoldre.

   En cada pas, l'algorisme ha d'inspeccionar q_i i q_{i-1} (1 ns), possiblement realitzar una suma o una resta (en el pitjor cas una resta en 10 ns), desplaçar els registres `S-HI-LO-X` un bit cap a la dreta (2 ns), escriure el nou valor en el registre `HI` (2 ns) i, finalment, avaluar el nombre de cicle actual (1 ns). La suma dels temps que costen aquestes operacions és de 16 ns.

2. Sent els operands de 16 bits, seran necessaris un total de 17 cicles per a realitzar la multiplicació: un cicle d'inicialització més 16 cicles per a processar els 16 bits. La durada de cada cicle s'ha determinat en l'apartat anterior (16 ns). El temps requerit per a realitzar una multiplicació serà de 17 cicles multiplicat per 16 ns, és a dir, 272 ns. La productivitat és la inversa del temps que costa cada operació, és a dir: $1/(272 \times 10^{-9}\ \text{s}) = 3676470.59$ operacions per segon $\simeq 3.68$ MOPS.

3. Tenint en compte que el temps de cicle està expressat en nanosegons i que s'ha d'expressar el resultat en MOPS, l'expressió de la productivitat en funció del nombre de bits és:

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

   El factor $n+1$ representa el nombre de cicles necessaris per a realitzar una operació.

■

PROBLEMA 19 Considerant nombres de 4 bits, realitzeu les multiplicacions que s'indiquen per mitjà del mètode de Booth, aplicant l'operador seqüencial i l'algorisme corresponents:

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

SOLUCIÓ Per a tots els casos, en primer lloc s'ha d'obtenir la codificació binària en complement a dos de cada un dels operands. En el pas d'inicialització, el registre M es carrega amb el valor del multiplicand, el multiplicador s'emmagatzema en el registre LO i la resta de bits (bit S, registre HI i bit auxiliar X) s'inicialitzen a zero.

En cada pas de l'algorisme es comença per analitzar el bit auxiliar (X) i el bit de menor pes del multiplicador (LO_0) per a determinar l'operació que s'ha de fer en aquest pas: sumar M a HI si $LO_0|X = 01$ o bé restar M a HI si $LO_0|X = 10$. Al final de cada pas, sempre s'ha de desplaçar el valor contingut en els registres S-HI-LO-X una posició cap a la dreta. Aquest desplaçament cap a la dreta és de tipus aritmètic, és a dir, amb còpia del bit de signe S sobre si mateix, a fi de preservar el signe del valor emmagatzemat en S-HI.

Es detallen a continuació els passos de l'algorisme per a cadascun dels apartats de l'enunciat. Els cicles es numeren des del 0 (cicle d'inicialització) fins al 4, ja que són 4 els bits que es representen els operands. El resultat, de 8 bits, queda emmagatzemat en els registres HI i LO en finalitzar l'últim pas de l'algorisme.

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

Cicle	Acció	S-HI-LO-X
0	Inicialització	0-0000-1111-0
1	$LO_0\|X = 10 \rightarrow$ Restar M	1-1101-1111-0
1	Desplaçar	1-1110-1111-1
2	$LO_0\|X = 11 \rightarrow$ Només desplaçar	1-1111-0111-1
3	$LO_0\|X = 11 \rightarrow$ Només desplaçar	1-1111-1011-1
4	$LO_0\|X = 11 \rightarrow$ Només desplaçar	1-1111-1101-1

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

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

Cicle	Acció	S-HI-LO-X
0	Inicialització	0-0000-0011-0
1	$LO_0\|X = 10 \rightarrow$ Restar M	1-1011-0011-0
1	Desplaçar	1-1101-1001-1
2	$LO_0\|X = 11 \rightarrow$ Només desplaçar	1-1110-1100-1
3	$LO_0\|X = 01 \rightarrow$ Sumar M	0-0011-1100-1
3	Desplaçar	0-0001-1110-0
4	$LO_0\|X = 00 \rightarrow$ Només desplaçar	0-0000-1111-0

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

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

Cicle	Acció	S-HI-LO-X
0	Inicialització	0-0000-1011-0
1	$LO_0\|X = 10 \rightarrow$ Restar M	0-0111-1011-0
1	Desplaçar	0-0011-1101-1
2	$LO_0\|X = 11 \rightarrow$ Només desplaçar	0-0001-1110-1
3	$LO_0\|X = 01 \rightarrow$ Sumar M	1-1010-1110-1
3	Desplaçar	1-1101-0111-0
4	$LO_0\|X = 10 \rightarrow$ Restar M	0-0100-0111-0
4	Desplaçar	0-0010-0011-1

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

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

Cicle	Acció	S-HI-LO-X
0	Inicialització	0-0000-0110-0
1	$LO_0 \mid X = 00 \rightarrow$ Només desplaçar	0-0000-0011-0
2	$LO_0 \mid X = 10 \rightarrow$ Restar M	0-0100-0011-0
2	Desplaçar	0-0010-0001-1
3	$LO_0 \mid X = 11 \rightarrow$ Només desplaçar	0-0001-0000-1
4	$LO_0 \mid X = 01 \rightarrow$ Sumar M	1-1101-0000-1
4	Desplaçar	1-1110-1000-0

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

■

PROBLEMA 20 Obteniu la recodificació per parelles de bits dels nombres següents, expressats en binari:

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

SOLUCIÓ S'ha de suposar l'existència d'un bit auxiliar, el valor del qual és zero i que està situat a la dreta del bit de menor pes. A partir d'ací, ha de procedir-se aplicant la taula de recodificació per parelles, prenent els bits de tres en tres des del bit auxiliar fins a arribar al de major pes. En el primer pas es consideren el bit auxiliar, el de pes 2^0 i el de pes 2^1 per a obtenir el dígit de recodificació per parelles de pes 2^0; En el segon pas es consideren els bits de pes 2^1, 2^2 i 2^3 per a obtenir el dígit de pes 2^2; en el tercer el 2^3, el 2^4 i el 2^5 per a obtenir el dígit de pes 2^4 i així successivament. Noteu que en la recodificació per parelles de bits s'obté només un dígit per cada dos dígits del nombre original.

La taula de recodificació per parelles de bits s'adjunta a continuació:

q_{i+1}	q_i	q_{i-1}	Recodificació
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

També és possible recodificar per parelles a partir d'una recodificació prèvia de Booth. En aquest cas, simplement es prenen els dígits de Booth de dos en dos des del de menor pes (sense necessitat de considerar un dígit auxiliar) i es transformen cada dos dígits de Booth en un dígit de parelles segons el valor que aporten els dos dígits de Booth a la quantitat representada pel nombre.

Les solucions de cada un dels casos de l'enunciat es presenten a continuació:

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 Indiqueu a quines quantitats (en decimal) corresponen els nombres següents recodificats per parelles:

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`

SOLUCIÓ Per a obtenir la quantitat representada per un nombre recodificat per parelles, ha de considerar-se que aquest nombre està representat en un sistema posicional de pesos 1, 2, 4, 8, etc. Cada dígit del nombre representa el seu propi valor (0, 1, 2, -1 o -2) multiplicat per 2 elevat a la potència que li correspon segons la seua posició en el nombre.

Els resultats per als nombres plantejats en l'enunciat són els següents:

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 Obteniu els productes següents pel mètode de recodificació per parelles de bits del multiplicador, suposant que s'utilitza l'operador seqüencial corresponent. Considereu operands de 6 bits.

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

SOLUCIÓ Com que són 6 els bits dels operands, seran necessaris $6/2 = 3$ passos de l'algorisme, més el pas corresponent a la inicialització de registres. En total quatre cicles de rellotge. En cada cicle i s'analitzaran els bits q_{i+1}, q_i i q_{i-1} del multiplicador, fent ús d'un bit extra (X) que representa a q_{i-1}. Les operacions a realitzar depenen del valor d'aquests tres bits, podent ser sumar o restar M o 2M a HI o bé deixar-lo intacte (en total, 5 possibilitats). Al final de cada cicle s'ha de realitzar un desplaçament aritmètic de S-HI-LO-X de dos posicions cap a la dreta.

Es presenten a continuació els resultats per a cadascun dels casos de l'enunciat.

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

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

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

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

Cicle	Acció	S-HI-LO-X
0	Inicialització	0-000000-001101-0
1	$LO_{1,0}\|X = 010 \rightarrow$ Sumar M	0-011001-001101-0
1	Desplaçar dues posicions a la dreta	0-000110-010011-0
2	$LO_{1,0}\|X = 110 \rightarrow$ Restar M	1-101101-010011-0
2	Desplaçar dues posicions a la dreta	1-111011-010100-1
3	$LO_{1,0}\|X = 001 \rightarrow$ Sumar M	0-010100-010100-1
3	Desplaçar dues posicions a la dreta	0-000101-000101-0

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

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

Cicle	Acció	S-HI-LO-X
0	Inicialització	0-000000-101101-0
1	$LO_{1,0}\|X = 010 \rightarrow$ Sumar M	1-101111-101101-0
1	Desplaçar dues posicions a la dreta	1-111011-111011-0
2	$LO_{1,0}\|X = 110 \rightarrow$ Restar M	0-001100-111011-0
2	Desplaçar dues posicions a la dreta	0-000011-001110-1
3	$LO_{1,0}\|X = 101 \rightarrow$ Restar M	0-010100-001110-1
3	Desplaçar dues posicions a la dreta	0-000101-000011-1

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

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

Cicle	Acció	S-HI-LO-X
0	Inicialització	0-000000-010110-0
1	$LO_{1,0}\|X = 100 \rightarrow$ Restar 2M	0-101000-010110-0
1	Desplaçar dues posicions a la dreta	0-001010-000101-1
2	$LO_{1,0}\|X = 011 \rightarrow$ Sumar 2M	1-100010-000101-1
2	Desplaçar dues posicions a la dreta	1-111000-100001-0
3	$LO_{1,0}\|X = 010 \rightarrow$ Sumar M	1-100100-100001-0
3	Desplaçar dues posicions a la dreta	1-111001-001000-0

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

■

PROBLEMA 23 Considereu l'operador de multiplicació seqüencial amb recodificació per parelles de bits per a nombres amb signe i el seu algorisme corresponent. Supose que el cost de les diferents operacions involucrades en l'algorisme és el següent:

- Inicialitzar registres i circuit de control: 2 ns.
- Inspeccionar q_{i+1}, q_i i q_{i-1}: 1 ns.
- Sumar: 9 ns.
- Restar: 10 ns.
- Desplaçar `M`: 2 ns.

- Desplaçar `S-HI-LO-X` 2 posicions: 3 ns.
- Escriure registre `HI`: 2 ns.
- Avaluar el numero de cicle actual: 1 ns.

Suposeu a més que el pas d'inicialització dels registres i del circuit de control es realitza en un cicle de rellotge a banda, abans de començar a processar els bits del multiplicador.

1. Quin és el mínim període aplicable al senyal de rellotge?
2. Si es consideren operands de 16 bits, quin és la productivitat (en MOPS) que pot proporcionar el multiplicador seqüencial?
3. Quina és l'expressió general de la productivitat del circuit expressada en MOPS en funció del nombre de bits n?

SOLUCIÓ

1. De forma semblant a com s'ha abordat el problema 18, per a determinar la duració mínima possible del cicle de rellotge és necessari considerar el retard de les operacions que han de realitzar-se en cada cicle. En el cas que hi haja diverses alternatives –amb aquest algorisme ocorre, ja que és possible haver de fer una suma, una resta, o suma o resta amb desplaçament o cap operació, segons el valor dels bits q_{i+1}, q_i i q_{i-1}–, es considerarà l'operació amb major retard, ja que el cicle de rellotge ha d'adequar-se al cas que coste més temps resoldre.

   En cada pas, l'algorisme ha d'inspeccionar q_{i+1}, q_i i q_{i-1} (1 ns), possiblement realitzar un desplaçament a l'esquerra del multiplicant (per a obtenir 2M, en 2 ns) i una suma o una resta (en el pitjor cas una resta en 10 ns), desplaçar els registres `S-HI-LO-X` dues posicions cap a la dreta (3 ns), escriure el nou valor en el registre `HI` (2 ns) i, finalment, avaluar el nombre de cicle actual (1 ns). La suma del que costa cadascuna d'aquestes operacions és de 19 ns.

2. Sent els operands de 16 bits, caldrà un total de 9 cicles per a fer la multiplicació: un cicle d'inicialització més $\frac{16}{2} = 8$ cicles per a processar els 16 bits, ja que en cada cicle es processarà un dígit que representa la recodificació de dos bits del multiplicador. La durada de cada cicle s'ha determinat en l'apartat anterior (19 ns). El temps requerit per a fer una multiplicació serà de 9 cicles multiplicat per 19 ns, és a dir, 171 ns. La productivitat és la inversa del temps que costa cada operació, és a dir: $1/(171 \times 10^{-9}\ \text{s}) = 5847953.22$ operacions per segon $\simeq 5.85$ MOPS.

3. Tenint en compte que el temps de cicle està expressat en nanosegons i que s'ha d'expressar el resultat en MOPS, l'expressió de la productivitat en funció del nombre 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 nombre de cicles necessaris per a fer una operació de multiplicació completa.

■
