# Apunts del Tema 5
# Aritmètica d'enters i coma flotant

Joan Manuel Parcerisa

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Març 2020

# Tema 4. Aritmètica d'enters i coma flotant

En aquest tema aprofundirem alguns aspectes de l'aritmètica de nombres enters i presentarem els nombres en coma flotant, així com el suport que ofereix l'ISA de MIPS en cada cas. En el cas dels enters, s'estudiaran les condicions de sobreeiximent (overflow) i les excepcions que se'n poden derivar, així com els algorismes de multiplicació i divisió, per als quals es proposaran realitzacions hardware senzilles. Per al cas de les dades en coma flotant s'estudiarà la seva representació en el format IEEE-754, incloent les codificacions especials, i s'estudiaran els algorismes de suma/resta i de multiplicació/divisió, així com l'arrodoniment dels resultats i el consegüent error de precisió. Els requadres en verd i l'apèndix final són extensions del temari opcionals.

3.2, 3.8(b)

## 1. Overflow de suma i resta d'enters

### 1.1 Definició

El rang de representació de nombres enters en Ca2 amb n bits és $R = [-2^{n-1}, 2^{n-1}-1]$. Diem que es produeix sobreeiximent o *overflow* en una operació quan el resultat exacte no pertany a aquest rang. I llavors el resultat calculat amb n bits no és correcte.

En el cas de la suma, es produeix sobreeiximent "si i sols si els operands són del mateix signe, i el resultat és de signe diferent".

Demostració

Considerem la suma exacta $s = a + b$, on $a$ i $b$ són enters i estan representats per les cadenes de bits $A=A_{n-1}, ..., A_0$ i $B=B_{n-1}, ..., B_0$. Sabem que la suma exacta $s$ sempre es pot representar amb n+1 bits com $S=S_n, S_{n-1}, ..., S_0$. ¿En quins casos la cadena truncada $S'=S_{n-1}, ..., S_0$ (prescindint del bit $S_n$) també és la representació correcta de $s$?

En primer lloc, suposem que $a$ i $b$ són de signes oposats (per exemple, $a$ és negatiu i $b$ positiu). Si $a < 0$, llavors $a + b < b$. Si $0 \le b$, llavors $a \le a + b$. Combinant les dues desigualtats, llavors $a \le a + b < b$, de manera que $s$ sempre es pot representar amb n bits, al igual que $a$ i $b$. S' és sempre la representació correcta de $s$.

En segon lloc, suposem que $a$ i $b$ són del mateix signe. Com que la suma exacta ha de ser del mateix signe que els operands, llavors $S_n=A_{n-1}=B_{n-1}$. La suma truncada S' representa l'enter $s$ si i només si S és l'extensió de signe de S', és a dir si $S_{n-1}=S_n$, i per tant si $S_{n-1}=A_{n-1}=B_{n-1}$. En altres paraules, S' és la representació correcta de $s$ si i sols si S' és del mateix signe que A i B.

En el cas de la diferència $d=a-b$, el resultat serà correcte si i només si compleix $a=d+b$. Aplicant la regla de l'overflow de la suma a aquesta expressió deduïm que es produeix sobreeiximent "si i sols si la diferència ($d$) és del mateix signe que el substraend ($b$) però de signe diferent que el minuend ($a$)".

## 1.2 Detecció de l'overflow en Ca2

En el curs anterior es va estudiar el disseny d'un sumador "amb propagació de carry", format per una cadena de n full-adders. Cada full-adder rep tres bits d'entrada: un bit de cada operand ($a_i$ i $b_i$) i un bit de "carry-in" ($c_i$) procedent del full-adder anterior. El full-adder calcula dos bits de sortida: un bit de suma $s_i=a_i \oplus b_i \oplus c_i$ i un bit de "carry-out" $c_{i+1} = (a_i \oplus b_i) \wedge c_i \vee a_i \wedge b_i$ que es connectarà al full-adder següent. En particular, per al darrer full-adder (bit de més pes) el carry-in és $c_i = c_{n-1}$ i el carry-out és $c_{i+1} = c_n$. Segons que es va estudiar en el curs anterior, es produeix overflow de la suma quan $c_{n-1} \neq c_n$. La detecció pel hardware de l'overflow d'enters en Ca2 (denotat pel símbol $v$) requereix doncs una simple porta lògica afegida al darrer full-adder: $v = c_{n-1} \oplus c_n$.

La suma (o resta) de nombres enters en complement a 2 té la propietat de seguir el mateix algorisme que la suma (o resta) de nombres naturals, i permet utilitzar el mateix hardware. No obstant, la detecció d'overflow és diferent per a enters ($v$) que per a naturals ($c_n$), com ja hem vist. Alguns llenguatges d'alt nivell com Fortran o Ada requereixen fer una distinció entre la suma o resta d'enters i la de naturals: en el cas que una suma o resta d'enters produeixi overflow, el programa ha de causar una excepció[1], mentre que els overflows de naturals han de ser ignorats. Això requereix tenir dos tipus d'instruccions: per a enters usarem `add`, `addi`, i `sub`, les quals causen excepció en cas d'overflow; i per a naturals usarem `addu`, `addiu` i `subu`, les quals ignoren els overflows.

En canvi, altres llenguatges d'alt nivell com C estipulen que tots els overflows de suma o resta siguin ignorats, ja siguin d'enters o de naturals. Així doncs, les sumes o restes d'enters i les de naturals es traduiran a assemblador usant les mateixes instruccions: `addu`, `addiu` i `subu`.

Pot succeir que un programa en C o en assemblador necessiti calcular si una suma ha produït overflow. Alguns processadors disposen d'instruccions específiques que permeten consultar en el hardware si tal condició s'ha produït. MIPS no ofereix aquesta informació però es pot calcular per software amb una senzilla seqüència d'instruccions, aplicant la definició. Suposant la suma d'enters de 32 bits $s = a+b$, l'overflow $v$ és:

$$v = (\overline{a_{31} \oplus b_{31}}) \wedge (a_{31} \oplus s_{31})$$

EXEMPLE 1: Suposem que els enters $a$, $b$, $s$ estan guardats en `$t0`, `$t1`, `$t2` respectivament, i executem `addu $t2,$t0,$t1`. Escriure una seqüència d'instruccions que calculi en `$t3` la condició d'overflow: 0 si la suma és correcta; 1 si hi ha hagut overflow.

```
xor    $t3, $t0, $t1       # a xor b
nor    $t3, $t3, $zero
xor    $t4, $t0, $t2       # a xor s
and    $t3, $t3, $t4
srl    $t3, $t3, 31        # trasllada el bit 31 a la posició 0
```

1. Quan un programa causa una excepció, la instrucció causant no escriu cap resultat en el registre destinació ni s'incrementa el registre PC. En comptes d'això, el programa salta immediatament a una rutina específica del sistema operatiu per al tractament d'aquesta situació. Al Tema 8 es veurà amb més detall una descripció del mecanisme d'excepcions.

3.3

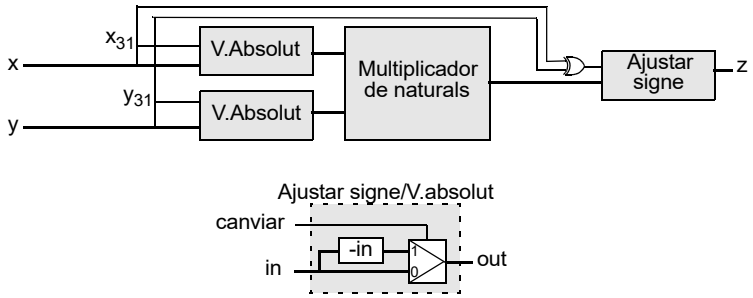
# 2. Multiplicació entera de 32 bits amb resultat de 64 bits

## 2.1 Algorisme de multiplicació entera

Estudiarem un algorisme de multiplicació entera basat en la multiplicació de nombres naturals anàleg al que apliquem quan ho fem manualment. Per exemple, per multiplicar (+3) × (-4) solem multiplicar els valors absoluts: 3 × 4 = 12, i posteriorment canviem el signe del resultat si els operands són de diferent signe: -12. L'algorisme per a calcular el producte z = x × y és doncs:

1. Calcular els valors absoluts
2. Multiplicar els valors absoluts (producte de naturals)
3. Canviar el signe del resultat si els operands tenen signes diferents

El hardware pot consistir en:




## 2.2 Algorisme tradicional de multiplicació de naturals

EXEMPLE 2: Vegem l'algorisme tradicional, amb números decimals de 3 dígits:

```
     348    multiplicand
×    951    multiplicador
------
     348    =   348 × 1
    1740    =  3480 × 5
+  3132     = 34800 × 9
------
  330948
```

Observem que es tracta d'obtenir primer tants productes parcials com dígits té el multiplicador (en cada producte multipliquem un dígit, ponderat per la corresponent potència de 10) i al final cal sumar tots els productes parcials.

EXEMPLE 3: Un exemple similar, amb números en base 2 de 4 bits:

```
  10          1010     multiplicand
× 13        ×  1101    multiplicador
------      ------
              1010   =    1010 × 1
             0000    =   10100 × 0
            1010     =  101000 × 1
           + 1010    = 1010000 × 1
= 130       10000010
```

Observem que en base 2 sols multipliquem per 1 (o per zero), i per tant els productes parcials es redueixen a una decisió binària en funció del bit del multiplicador: sumar o no sumar el multiplicand, desplaçat segons el pes específic del bit del multiplicador.

## 2.3 Circuit seqüencial per a la multiplicació (z = x × y) de naturals de 32 bits

Estudiarem un circuit seqüencial per implementar l'anterior algorisme. A fi d'estalviar components de hardware, el modificarem una mica: en lloc de calcular primer tots els productes parcials i sumar-los al final (caldrien múltiples registres per guardar-los), els anirem sumant a mesura que els calculem, de manera que el resultat de cada pas se suma a un únic registre acumulador P de 2·n bits (en groc), que s'inicialitza a zero i acaba contenint el Producte final:

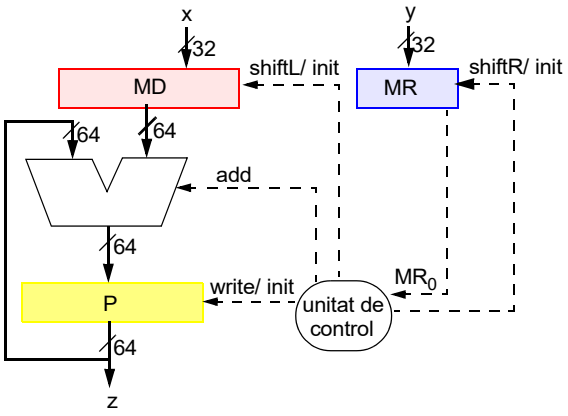
```
      0000   P (inicial)
+     1010   =    1010 × 1
      1010   P
+    00000   =   10100 × 0
     01010   P
+   101000   =  101000 × 1
     110010   P
+  1010000   = 1010000 × 1
  10000010   P
```

MD= 00001010
P= 00000000
MR= 1101

El multiplicand (**x = 1010** en l'exemple) s'emmagatzema al registre MD (en vermell) i el multiplicador (**y = 1101** en l'exemple) en el registre MR (en blau). En cada iteració s'obté un producte parcial multiplicant un bit de MR (començant pel de menor pes) pel registre MD desplaçat a l'esquerra tantes posicions com iteracions completades, a fi de considerar el pes específic del bit de MR que multipliquem. A fi de recórrer els bits de MR, el nostre circuit llegirà sempre el bit de menor pes i a continuació desplaçarà tot el registre una posició a la dreta. A fi d'obtenir el valor de MD desplaçat a l'esquerra tantes posicions com iteracions completades, el nostre circuit el desplaçarà a l'esquerra una posició després de cada iteració. Observem doncs que la llargada de MD ha de ser de 2·n bits, i que s'ha de inicialitzar amb zeros a la part alta i amb el multiplicand **x** a la part baixa.

A continuació es mostra el circuit per a n=32 bits, on els blocs seqüencials MD, MR i P no són mers registres, sinó que contenen també la lògica combinacional necessària per a realitzar les operacions indicades pels senyals de la Unitat de Control: inicialització (Init), desplaçaments (ShiftL, ShiftR) i càrrega (Write). A la dreta es mostra l'algorisme que descriu el funcionament de l'autòmata, en pseudocodi. Els subíndexs denoten rangs de bits. Els operadors >> i << denoten desplaçaments a esquerra i dreta respectivament.

Multiplicador seqüencial de naturals: z = x * y




Pseudocodi

```
// Inicialització
MD0:31 = x; MD32:63 = 0;
P  = 0;
MR = y;

for (i=1; i<=32; i++)
{
      if (MR0 == 1)
            P = P + MD;
      MD = MD << 1;
      MR = MR >> 1;
}
z = P;
```

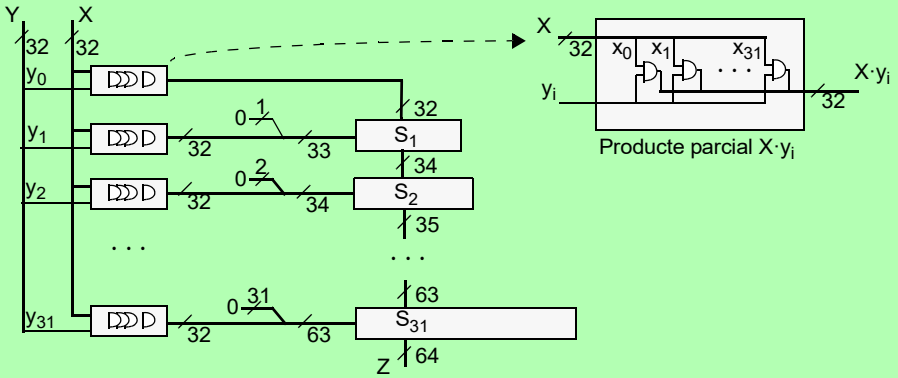
Aquest circuit seqüencial tarda un mínim de 33 cicles a obtenir el producte, i el temps de cicle ha de ser prou llarg per a completar una suma de 64 bits.

EXEMPLE 4: La següent taula fa el seguiment del producte amb 4 bits per a x=**1010**$_2$, y=**1101**$_2$. Cada fila mostra el contingut dels tres registres P, MD i R després de la inicialització i al final de cada una de les quatre iteracions.

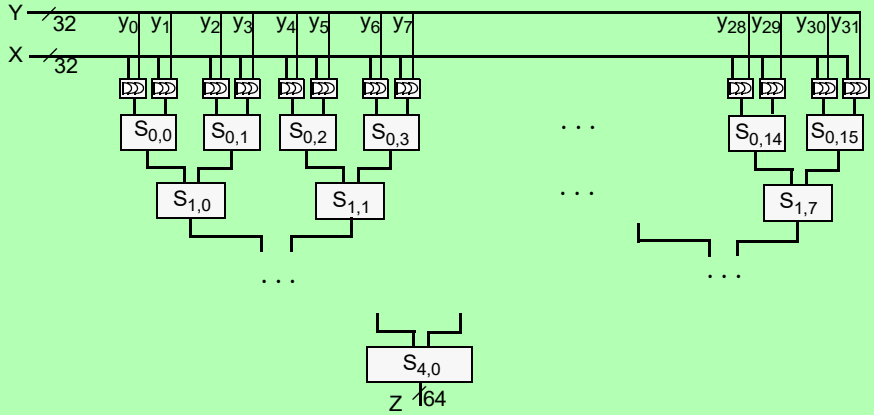
| iter. | P (Producte) | | | | | | | | MD (Multiplicand) | | | | | | | | MR (Multiplicador) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| inicial | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **0** | **1** | **0** | **1** | **1** | **0** | **1** |
| 1 | 0 | 0 | 0 | **0** | **1** | **0** | **1** | **0** | 0 | 0 | 0 | **1** | **0** | **1** | **0** | 0 | 0 | **1** | **1** | **0** |
| 2 | 0 | 0 | **0** | **0** | **1** | **0** | **1** | **0** | 0 | 0 | **1** | **0** | **1** | **0** | 0 | 0 | 0 | 0 | **1** | **1** |
| 3 | 0 | **0** | **1** | **1** | **0** | **0** | **1** | **0** | 0 | **1** | **0** | **1** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | **1** |
| 4 | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **1** | **0** | **1** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Circuit multiplicador combinacional, més ràpid

La multiplicació de 32 bits es podria fer també amb un circuit combinacional que calculi: $X \cdot y_0 + (X \cdot y_1)<<1 + (X \cdot y_2)<<2 + \ldots + (X \cdot y_{31})<<31$. És fàcil veure que caldran 31 sumadors disposats en sèrie, des de $S_1$ (de 33 bits) fins a $S_{31}$ (de 63 bits). El darrer sumador $S_{31}$ produeix 63 bits de suma i un bit de carry, és a dir un resultat Z de 64 bits en total:




El retard del circuit és aproximadament la suma dels retards dels 31 sumadors. Per tant, no representa una millora substancial respecte del sumador seqüencial. Però es poden fer moltes sumes en paral·lel si reestructurem el circuit en forma d'arbre. Llavors, el retard és aproximadament la suma dels retards de 5 sumadors, 6 cops més ràpid que l'anterior



### 2.4 Multiplicació d'enters i de naturals en MIPS

L'ISA de MIPS disposa de dues instruccions, `mult` i `multu`, per a multiplicar enters i naturals, respectivament.

```
mult   rs, rt        # $hi:$lo <- rs * rt (enters)
multu  rs, rt        # $hi:$lo <- rs * rt (naturals)
```

Les dues instruccions calculen el resultat exacte amb 64 bits, i escriuen la part alta del resultat en el registre `$hi`, i la part baixa en el registre `$lo`. Aquests dos registres són implícits i no poden ser usats com a operands explícits per cap instrucció. Per a obtenir-ne el valor cal copiar-lo a un registre normal per mitjà de les instruccions `mfhi` i `mflo`:

```
mfhi   rd            # rd <- $hi
mflo   rd            # rd <- $lo
```

El producte de nombres de 32 bits es defineix en C, i en altres llenguatges d'alt nivell, com una operació amb un resultat de 32 bits, i per tant es tradueix ignorant la part alta del resultat. Aquest resultat truncat pot no ser representable amb 32 bits (overflow), és a dir que el resultat amb 32 bits (`$lo`) no representa el mateix número que amb 64 bits (`$hi:$lo`). Per detectar-lo, cal examinar les següents condicions:

- `multu` causa overflow de naturals quan `$hi` és diferent de zero.
- `mult` causa overflow d'enters quan `$hi:$lo` no és l'extensió de signe de `$lo`.

3.4, 3.8(a)

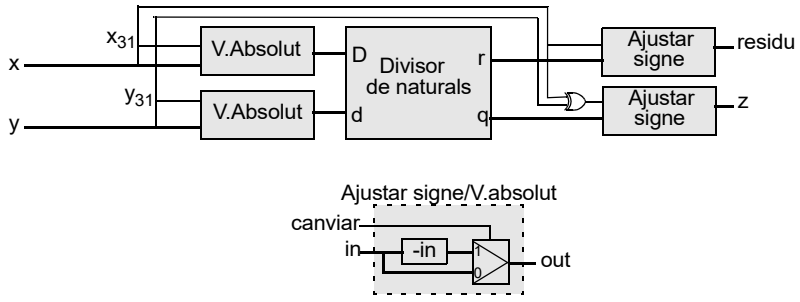
## 3. Divisió entera de 32 bits amb càlcul del residu

### 3.1 Algorisme de divisió entera

Anàlogament a la multiplicació, estudiarem un algorisme de divisió entera basat en la divisió de nombres naturals. L'algorisme de la divisió z = x / y és doncs:

1. Calcular els valors absoluts
2. Dividir els valors absoluts (divisió de naturals) amb càlcul de quocient i residu
3. Canviar el signe del quocient si els operands tenen signes diferents, i el del residu, si el dividend és negatiu.

L'esquema del hardware és també similar al de la multiplicació:



### 3.2 Algorisme tradicional de divisió de naturals

La divisió de naturals del dividend D entre el divisor d consisteix a trobar el quocient q i el residu r naturals tals que compleixin $D = d \cdot q + r$, essent $r<q$.

EXEMPLE 5: Vegem l'algorisme tradicional, amb números decimals de 3 dígits (amb valors entre 0 i 9). Volem dividir el dividend D=421 entre el divisor d=013. L'algorisme consisteix a buscar el tres dígits $q_2$, $q_1$, $q_0$ del quocient, i el residu r.

Com que $q = q_2 \cdot 10^2 + q_1 \cdot 10^1 + q_0 \cdot 10^0$, podem reescriure la igualtat:

$$421 = 013 \cdot (q_2 \cdot 10^2 + q_1 \cdot 10^1 + q_0 \cdot 10^0) + r$$
$$421 = q_2 \cdot 013 \cdot 10^2 + q_1 \cdot 013 \cdot 10^1 + q_0 \cdot 013 \cdot 10^0 + r$$
$$421 = q_2 \cdot \mathbf{013}00 + q_1 \cdot \mathbf{013}0 + q_0 \cdot \mathbf{013} + r$$

El dígit $q_2$ s'obté buscant el major dígit tal que $421 \geq q_2 \cdot \mathbf{013}00$. És $q_2=0$. Ara restarem $q_2 \cdot \mathbf{013}00$ (= **000**00) als dos termes de la igualtat: 421-**000**00=421. I quedarà:

$$421 = q_1 \cdot \mathbf{013}0 + q_0 \cdot \mathbf{013} + r$$

El dígit $q_1$ s'obté d'igual manera: el major dígit tal que $421 \geq q_1 \cdot \mathbf{013}0$. És $q_1=3$. Ara restarem $q_1 \cdot \mathbf{013}0$ (= **039**0) als dos termes de la igualtat: 421-**039**0=031. I quedarà:

$$031 = q_0 \cdot \mathbf{013} + r$$

El dígit $q_0$ s'obté d'igual manera: el major dígit tal que $031 \geq q_0 \cdot \mathbf{013}$. És $q_0=2$. Llavors restarem $q_0 \cdot \mathbf{013}$ (=**026**), als dos termes de la igualtat: 031-**026**=005. I quedarà:

$$005 = r$$

Típicament, aquest algorisme el denotem de la següent manera:

**1ª iteració:**

```
    421  | 013
- 00000  | 0
    421
```

com que 421 > **013**00, trobem $q_2$=**0**
li restem **0** × **013**00 = **000**00
el residu queda: 421

**2ª iteració:**

```
    421  | 013
    421  | 03
- 0390
    031
```

com que 421 ≥ 3 × 0130, trobem $q_1$=**3**
li restem **3** × **013**0 = **039**0
el residu queda: 031

**3ª iteració:**

```
    421  | 013
    421  | 032
    031
- 026
    005
```

com que 031 ≥ 2 × **013**, trobem $q_0$=**2**
li restem **2** × **013** = **026**
el residu queda: 005

EXEMPLE 6: Vegem un exemple similar, amb números en base 2 de 4 bits. Volem dividir el dividend D=1011 entre el divisor d=0010. Podem escriure la igualtat:

$$1011 = 0010 \cdot (q_3 \cdot 2^3 + q_2 \cdot 2^2 + q_1 \cdot 2^1 + q_0 \cdot 2^0) + r$$
$$1011 = q_3 \cdot \mathbf{0010}000 + q_2 \cdot \mathbf{0010}00 + q_1 \cdot \mathbf{0010}0 + q_0 \cdot \mathbf{0010} + r$$

El procediment és similar:

El dígit $q_3$ s'obté buscant $q_3$ tal que $1011 \geq q_3 \cdot \mathbf{0010}000$. Com que $q_3$ sols pot ser 1 o 0, es tracta simplement de comparar si $1011 \geq \mathbf{0010}000$. És una simple comparació, que es pot realitzar amb una resta i observant si el resultat produeix o no un *carry-out* o *borrow*. En aquest cas, el resultat és fals, i trobem $q3=0$. Ara restarem $q_3 \cdot \mathbf{0010}000$ (= $\mathbf{0000}000$) als dos termes de la igualtat: $1011-\mathbf{0000}000=1011$. I quedarà:

$$1011 = q_2 \cdot \mathbf{0010}00 + q_1 \cdot \mathbf{0010}0 + q_0 \cdot \mathbf{0010} + r$$

El dígit $q_2$ s'obté d'igual manera comparant si $1011 \geq \mathbf{0010}00$. És cert: $q_2=1$. Ara restarem $q_2 \cdot \mathbf{0010}00$ (= $\mathbf{0010}00$) als dos termes de la igualtat: $1011-\mathbf{0010}00=0011$. Queda:

$$0011 = q_1 \cdot \mathbf{0010}0 + q_0 \cdot \mathbf{0010} + r$$

El dígit $q_1$ s'obté d'igual manera comparant si $0011 \geq \mathbf{0010}0$. És fals: $q_1=0$. Ara restarem $q_1 \cdot \mathbf{0010}0$ (= $\mathbf{0000}0$) als dos termes de la igualtat: $0011-\mathbf{0000}0=0011$. I quedarà:

$$0011 = q_0 \cdot \mathbf{0010} + r$$

El dígit $q_0$ s'obté d'igual manera comparant si $0011 \geq \mathbf{0010}$. És cert: $q_0=1$. Ara restarem $q_0 \cdot \mathbf{0010}$ (= $\mathbf{0010}$) als dos termes de la igualtat: $0011-\mathbf{0010}=0001$. I quedarà:

$$0001 = r$$

Anàlogament al cas dels decimals, aquest algorisme el denotem així:

**1ª iteració:**

```
    1011  | 0010
  -0000000  0
    1011
```

com que $1011 \geq \mathbf{0010}000$ és fals, $q_3=\mathbf{0}$
li restem $0 \times \mathbf{0010}000 = \mathbf{0000}000$
el residu queda igual: $1011$

**2ª iteració:**

```
    1011  | 0010
    1011    01
  - 001000
    0011
```

com que $1011 \geq \mathbf{0010}00$ és cert, $q_2=\mathbf{1}$
li restem $1 \times \mathbf{0010}00 = \mathbf{0010}00$
el residu queda: $0011$

**3ª iteració:**

```
    1011  | 0010
    1011    010
    0011
  - 00000
    0011
```

com que $0011 \geq \mathbf{0010}0$ és fals, $q_1=\mathbf{0}$
li restem $0 \times \mathbf{0010}0 = \mathbf{0000}0$
el residu queda igual: $0011$

**4ª iteració:**

```
    1011  | 0010
    1011    0101
    0011
    0011
  - 0010
    0001
```

com que $0011 \geq \mathbf{0010}$ és cert, $q_0=\mathbf{1}$
li restem $1 \times \mathbf{0010} = \mathbf{0010}$
el residu queda: $0001$

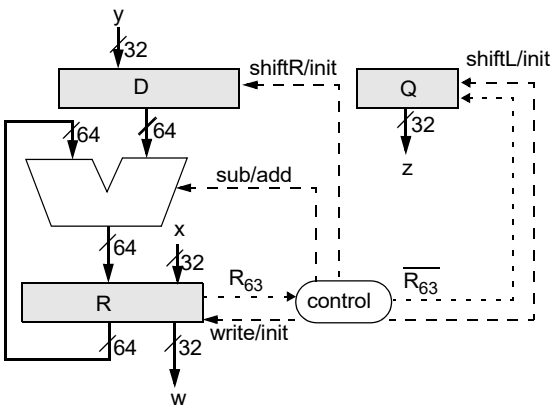
Resumint: Inicialment afegim 4 bits a la dreta del divisor (**0010**0000). En començar cada un dels 4 passos, desplacem el divisor 1 bit a la dreta (**0010**000, **0010**00, **0010**0 i **0010**) i comparem si el dividend és major o igual que aquest divisor restant-los i observant el bit de carry-out. Si hi ha carry-out, la comparació ha fallat i el bit del quocient és 0, en cas contrari és 1. Si el bit del quocient és 0, el dividend es queda igual. Si és 1, la multiplicació pel divisor és trivial i el nou dividend s'obté simplement de restar l'anterior menys el divisor. Però aquesta resta és precisament el resultat de la resta feta pel comparador, de manera que estalviem una operació.

### 3.3 Circuit seqüencial per a la divisió de naturals de 32 bits "amb restauració".

El circuit que proposem calcula el quocient ($z = x/y$) i el residu ($w = x\%y$) de la divisió de naturals de 32 bits seguint el mateix algorisme de l'exemple anterior. L'esquema següent representa el circuit divisor de naturals. L'ALU conté un sumador/restador, els registres R (dividend/residu) i D (divisor) tenen 64 bits, i el registre Q (quocient) té 32 bits. R s'inicialitza amb *x* a la part baixa i zeros a la part alta. D s'inicialitza amb *y* a la part alta i zeros a la part baixa, i Q s'inicialitza a zero. Al final de l'algorisme, Q conté el quocient *z,* i la part baixa de R conté el residu *w.*

El dividend que s'obté després de cada pas pot quedar igual que estava o bé ser el resultat de restar-li el divisor, però això no se sap fins després de la comparació, de manera que caldria guardar els dos valors. Per tal d'estalviar un registre, adoptarem l'algorisme anomenat "divisió amb restauració", que consisteix a escriure el resultat de la comparació (que es fa amb una resta) en el propi registre del dividend, el qual es perd. Aquest algorisme estalvia hardware però requereix més feina, ja que si la comparació falla llavors cal "restaurar" el dividend original tornant-li a sumar el divisor que abans se li ha restat.

**Divisor seqüencial de naturals**




**Pseudocodi**

```
// Inicialització
R63:32 = 0; R31:0 = x;
D63:32 = y; D31:0 = 0;
Q  = 0;
for (i=1; i<=32; i++) {
      D = D >> 1;
      R = R - D;
      if (R63 == 0)
            Q = (Q << 1) | 1;
      else {
            R = R + D;
            Q = Q << 1;
      }
}
```

Amb els operands naturals d'aquest algorisme, el resultat de la resta serà positiu si el primer és "major o igual que" el segon, o bé negatiu en cas contrari. El resultat de la comparació queda emmagatzemat al bit de signe $R_{63}$ i el seu negat és el bit de quocient a introduir a la posició 0 del registre Q, tot desplaçant els altres bits a l'esquerra.

EXEMPLE 7: La següent taula fa el seguiment de la divisió *x*/*y* amb 4 bits per a $x$=**1011**$_2$, $y$=**0010**$_2$ (com la del darrer exemple). Cada fila mostra el contingut dels tres registres després de la inicialització i al final de cada una de les quatre iteracions.

| iter. | R (Dividend/Residu) | | | | | | | | D (Divisor) | | | | | | | | Q (Quocient) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| init | 0 | 0 | 0 | 0 | **1** | **0** | **1** | **1** | **0** | **0** | **1** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | **1** | **0** | **1** | **1** | 0 | **0** | **0** | **1** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 2 | 0 | 0 | 0 | 0 | **0** | **0** | **1** | **1** | 0 | 0 | **0** | **0** | **1** | **0** | 0 | 0 | 0 | 0 | **0** | **1** |
| 3 | 0 | 0 | 0 | 0 | **0** | **0** | **1** | **1** | 0 | 0 | 0 | **0** | **0** | **1** | **0** | 0 | 0 | **0** | **1** | **0** |
| 4 | 0 | 0 | 0 | 0 | **0** | **0** | **0** | **1** | 0 | 0 | 0 | 0 | **0** | **0** | **1** | **0** | **0** | **1** | **0** | **1** |

## 3.4 Divisió d'enters i de naturals en MIPS

L'ISA de MIPS disposa de dues instruccions, `div` i `divu`, per a dividir enters i naturals, respectivament.

```
div    rs, rt        # $lo <- rs/rt, $hi <- rs%rt    (enters)
divu   rs, rt        # $lo <- rs/rt, $hi <- rs%rt    (naturals)
```

Les dues instruccions calculen el resultat exacte amb 64 bits, i escriuen el quocient en el registre `$lo`, i el residu en el registre `$hi`.

En cas que el divisor sigui zero, el resultat és indefinit (pot donar qualsevol resultat). Fora d'aquest cas, la divisió de naturals no pot donar mai un resultat no-representable (overflow). En canvi, la divisió d'enters pot donar overflow en un únic cas: si el dividend val $-2^{31}$ i el divisor val -1, ja que el resultat correcte $2^{31}$ no és representable en Ca2 amb 32 bits degut que el rang és asimètric (el major enter representable és $2^{31}-1$).

## 3.5 Divisió per potències de 2

L'ISA de MIPS disposa de la instrucció de desplaçament de bits a la dreta `srl` i `sra` que calculen el quocient de la divisió d'un natural o d'un enter, respectivament, per una potència de 2. Per al cas de naturals, `srl` dóna el mateix quocient que `divu`. Per al cas d'enters, si el dividend és positiu, tant `sra` com `div` donen el mateix quocient, però si aquest és negatiu i la divisió no és exacta, els resultats de `sra` i `div` són diferents.

EXEMPLE 8: Per veure la diferència, calculem el quocient $D/d$ que obtindríem amb la instruccció `div` i amb `sra`, suposant que dividim $D = -15$, entre $d = 4$. Amb `div -15,4` obtenim $q = -3$, i $r = -3$. En canvi, amb `sra -15,2` obtenim $q = -4$ (i per tant el residu, encara que no el calcula, és $r = 1$). Obtenim resultats diferents!

Això és així perquè `sra` i `div` obeeixen a definicions diferents de la divisió:

`sra`: Divideix $D/d$ calculant el parell de nombres $(q, r)$ que satisfan $D = d \times q + r$, tals que $0 \le |r| < |d|$ ($r$ és sempre positiu).

`div`: Divideix $D/d$ calculant el parell de nombres $(q, r)$ que satisfan $D = d \times q + r$, tals que $|r| < |d|$ i $r$ té el mateix signe que $D$.

Mentre que la definició de `sra` correspon a la divisió euclídea, la definició de `div` correspon a l'algorisme consistent a dividir valors absoluts i ajustar el signe dels resultats, tal com s'ha explicat en apartats anteriors, és a dir que compleix $|D| = |d| \times |q| + |r|$.

En general, les operacions de divisió (/) i mòdul (%) d'enters de 32 bits en llenguatges d'alt nivell com C es tradueixen sempre amb la instrucció `div` (o bé `divu`, si els operands estan declarats com `unsigned`).

3.5

# 4. Representació de nombres en coma flotant

La codificació en coma flotant serveix per representar nombres reals de forma aproximada amb un nombre limitat de bits. S'utilitza també per representar nombres tan grans com el nombre de segons d'un segle o tan petits com la càrrega elèctrica d'un electró.

## 4.1 Notació científica normalitzada (base 10)

Podem expressar el número 234000000 en forma exponencial substituint els zeros a la dreta per una potència de 10 així: $234\times10^6$. La segona notació no sols és més compacta, sinó que fa explícit que la seva precisió és de 3 dígits significatius. En canvi, la primera expressió és ambígua respecte de la seva precisió: no deixa clar si té 3 dígits significatius i els 6 zeros s'hi han afegit per denotar l'escala del 234 (milions), o bé és que el número és exacte fins a les unitats i per tant té 9 dígits significatius.

No obstant, el format exponencial no té una única codificació, ja que el mateix número el podríem expressar com $0{,}0234\times10^{10}$, com $2{,}34\times10^8$, etc. Per aquesta raó, en disciplines com l'enginyeria o les ciències s'acostuma a usar una notació estandaritzada, la "notació científica normalitzada". En aquesta notació, un número *v* s'escriu en la forma

$$v = m \times 10^n$$

On *n* (l'exponent) és un número enter i *m* (la mantissa) és un número amb una part entera *e* i una part fraccionària *f* separades per una coma, i amb la restricció que $0 < e \leq 9$ i que *f* tingui almenys 1 dígit. Així doncs, l'anterior exemple en notació científica normalitzada s'escriuria $2{,}34\times10^8$. En molts dispositius electrònics amb pantalles de baixa resolució s'acostuma a evitar el superíndex de l'exponent i se substitueixen els tres caràcters "×10" pel símbol "E" que separa mantissa i exponent: 2.34E8.

La notació científica normalitzada, ja coneguda per l'estudiant de cursos anteriors, és un cas particular de codificació en coma flotant. És un format que separa clarament la funcionalitat de la mantissa i de l'exponent: mentre que la mantissa expressa els dígits significatius de la magnitud, l'exponent indica el factor d'escala o, per entendre'ns, la posició de la coma. D'aquí ve el nom de "coma flotant", ja que la coma "es situa" més a la dreta o a l'esquerra segons el valor de l'exponent.

## 4.2 Representació binària

Els nombres en coma flotant en base 2 es representen segons el següent patró:

$$v = \pm \overbrace{1,xxx\ldots x}^{\text{fracció(F)}} \times 2^{\overbrace{yyy\cdots y}^{\text{exponent(E)}}}$$

signe(S) mantissa base

Essent *x*, *y* dígits binaris de la mantissa i l'exponent respectivament. A fi que el nombre estigui normalitzat, cal que la part entera de la mantissa tingui un sol bit i que sigui no nul: així doncs, el bit a l'esquerra de la coma sols pot ser un 1. A fi d'aprofitar al màxim la capacitat expressiva dels bits disponibles, aquest bit és implícit i no s'inclou explícitament en la representació codificada, però cal tenir-lo en compte a l'hora de fer càlculs: l'anomenem "bit ocult". Anàlogament, tampoc s'hi representa el valor de la base (és 2), ja que també és implícit.

D'una manera més formal, podríem dir que aquest format satisfà la següent igualtat:

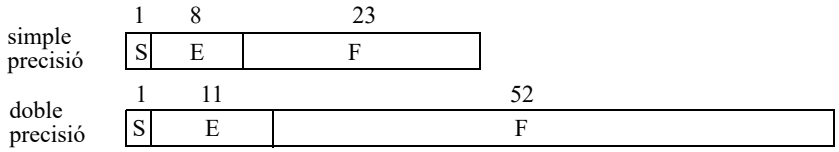
$$v = (-1)^s \times (1 + 0,f) \times 2^e$$

On *s*=S; *0,f* és la fracció representada per F, i *e* és l'enter representat per E.

## 4.3 L'estàndard IEEE-754

La representació de nombres en coma flotant en els computadors (en base 2) data des del mateix inici de la informàtica. Al principi cada fabricant va adoptar un format de coma flotant propi, amb alternatives diverses pel que fa al nombre de bits dedicats a codificar la mantissa o l'exponent, i en algun cas fins i tot adoptant la base 16 per a la potència en lloc de la base 2. No obstant, la necessitat d'intercanviar informació entre diferents sistemes va incentivar l'adopció d'un estàndard comú. El 1985 sortia a la llum l'estàndard IEEE-754 que no sols estableix el format de la representació, sinó que també en regula les operacions, l'arrodoniment, el tractament de les excepcions, etc. Posteriorment, l'estàndard s'ha anat modificant i ampliant en successives revisions (la darrera, el 2008).

L'estàndard constitueix un document voluminós que preveu múltiples opcions per tal de poder ser adoptat per arquitectures amb compromisos de cost i rendiment molt diversos, i tot i així ser perfectament compatibles entre si. En aquest curs n'estudiarem els aspectes més importants. Dels diversos formats definits, estudiarem solament el de "simple precisió" (32 bits) i el de "doble precisió" (64 bits), àmpliament utilitzats pels llenguatges d'alt nivell (corresponen als tipus de dades *float* i *double* del llenguatge C) :




### a) Signe (S)

0=positiu, 1=negatiu

### b) Fracció (F)

Part fraccionària de la mantissa. La mantissa sencera s'obté afegint-li el "bit ocult".

### c) Exponent (E)

Un dels objectius de l'estàndard és que les magnituds (prescindint del bit de signe), puguin ser comparades amb un comparador de naturals. Això requereix dues condicions:

- Primera, que els bits de l'exponent ocupin la part alta del número, a continuació del signe, i els de la fracció ocupin la part baixa. Així, serà major el nombre amb major exponent; però a iguals exponents, serà major el de major fracció.
- Segona, que l'exponent (un enter) es codifiqui en excés a 127 (en simple precisió) o a 1023 (en doble precisió). En aquest sistema de representació, si considerem totes les cadenes de bits i les ordenem de menor a major interpretades com enters, ja queden igualment ordenades interpretant-les com a naturals. En altres paraules, la correspondència entre els enters representables i els seus valors explícits és una funció monòtona creixent, cosa que no es compleix en Ca2.

El rang de valors representables per a l'exponent és $E \in [E_{min}, E_{max}]$

## 4.4 Overflow, rang i precisió

El rang de la representació ve definit per l'interval format pel major número positiu representable i el seu oposat. Aquest número es forma amb la màxima mantissa (tot els bits a 1) i el màxim exponent ($E_{max}$). L'extensió d'aquest rang depèn principalment del nombre de bits d'exponent. Si un resultat té un exponent $E > E_{max}$, aquest no és representable i es produeix una excepció d'*overflow*, que caldrà tractar adequadament.

Es defineix el concepte de precisió d'una representació com el seu nombre de bits significatius. La precisió expressa el grau de detall amb el qual es pot representar una quantitat. Matemàticament, per a un nombre amb una coma, es consideren significatius tots els dígits excepte els zeros a l'esquerra. Per tant, com que la mantissa de l'estàndard està normalitzada, la precisió és igual al nombre de bits totals de la mantissa (24 en simple precisió i 53 en doble precisió).

L'elecció d'un determinat format de coma flotant a partir d'un nombre fix de bits totals disponibles estableix un compromís entre precisió i rang, ja que augmentar la precisió (bits de mantissa) va en detriment del rang (bits d'exponent) i viceversa.

## 4.5 Error de precisió i underflow

L'error de precisió (també anomenat error de representació) és el que es comet pel fet de tenir la mantissa un nombre limitat de bits. Suposem que el resultat $v$ d'una operació no es pot representar de forma exacta amb la precisió donada, i que $v_0 < v < v_1$, essent $v_0$ i $v_1$ els valors representables més pròxims. Si donem el resultat aproximant-lo per $v_0$, l'error absolut de precisió comès es defineix com $\varepsilon = |v - v_0|$. El màxim error absolut en l'interval $(v_0,v_1)$ està afitat per la diferència entre els extrems

$$\varepsilon_{max} < |v_1 - v_0|$$

En simple precisió, podem calcular la fita d'error absolut. Donat un número qualsevol de la forma $v0 = m \times 2^E$, el següent valor representable és $v1 = (m+2^{-23}) \times 2^E$, i per tant

$$\varepsilon_{max} < (m + 2^{-23} - m) \times 2^E = 2^{E-23}$$

Sovint però, l'error absolut resulta poc significatiu. Per exemple, no té la mateixa importància un error d'un metre en la distància de la Terra al Sol que en la llargada d'un pont. És per això que ens interessa en realitat l'error relatiu de precisió $\eta = \varepsilon / |v|$. Podem trobar una fita a aquest error relatiu en l'interval $(v_0,v_1)$ considerant el màxim error absolut $\varepsilon_{max}$ i l'extrem de mínima magnitud de l'interval (suposant que $v>0$, aquest és $v_0$)

$$\eta_{max} < \varepsilon_{max} / |v_0|$$

En simple precisió podem calcular la fita d'error relatiu. Donat un número qualsevol de la forma $v0 = m \times 2^E$ una fita de l'error relatiu és

$$\eta_{max} < 2^{E-23} / m \times 2^E = 2^{-23} / m$$

i com que $m \geq 1.0$, resulta també que

$$\eta_{max} < 2^{-23} = 1 \text{ ULP}$$

Aquesta quantitat es coneix també com ULP (Unit in the Last Place) expressió que fa referència al número $2^{-23} = 0{,}0 \ldots 001$, que té un sol bit a 1 a la posició de menor pes.

L'error relatiu de precisió per a qualsevol valor en el format de simple precisió és sempre menor que $2^{-23}$, i aquesta fita depèn exclusivament de la precisió, és a dir del nombre de bits significatius de la mantissa.

No obstant, hi ha una excepció notable, i es produeix quan el valor absolut del resultat és menor que el mínim positiu representable $0 < |v| < 2^{Emin}$. Si l'aproximem simplement per 0, l'error absolut és $\varepsilon = |v - 0|$ i l'error relatiu és $\eta = \varepsilon / |v| = 1$. En altres paraules, l'error relatiu en el petit interval $(-2^{Emin}, 2^{Emin})$ és 6 ordres de magnitud major que a la resta del rang! La conclusió és que quan es manegen magnituds molt petites es poden obtenir resultats amb errors relatius molt importants que invaliden el seu significat.

Detectar quan es produeix un resultat en aquest rang de valors és important, i el hardware es pot configurar de manera que, en comptes d'arrodonir el resultat a 0, produeixi una excepció anomenada *Underflow*, per mitjà de la qual el software pot gestionar la situació amb un tractament oportú.

## 4.6 Codificacions especials

Durant la definició de l'estàndard es va considerar convenient reservar dues codificacions de l'exponent per a representar determinats valors especials. En concret, els valors E=0 ... 000 (tot zeros) i E=1 ... 111 (tot uns) queden reservats, i el rang d'exponents representables es redueix lleugerament. El mínim exponent té la forma $E_{min}$ = 0 ... 001 i el màxim té la forma $E_{max}$ = 1 ... 110, de manera que el rang en simple precisió és [-126, +127], i en doble precisió és [-1022, +1023].

### a) Zero

Els formats de coma flotant que hem estudiat fins ara són incapaços de representar el zero, ja que no hi ha cap fracció (F) ni exponent (E) que satisfaci la igualtat

$$0 = (-1)^S \times (1 + 0, F) \times 2^E$$

Així doncs, usarem un dels exponents reservats per a codificar el zero. Per convenció, el zero es representa amb tots els bits de fracció a zero i els d'exponent també:

F = 000 . . . 0, E = 000 . . .0

Degut al signe, el zero té dues representacions, cosa que complica les comparacions.

### b) Inf

L'infinit no és pròpiament un valor real, però és un concepte adoptat per raons pràctiques. Considerem per exemple l'expressió

$$y = \frac{1}{1 + \frac{100}{x}}$$

Per a valors de $x$ pròxims a zero, el quocient $100/x$ pot produir una excepció d'overflow que causi un error en el programa. No obstant, sabem que existeix un valor finit que aproxima el resultat del càlcul, ja que

$$\lim_{x \to 0} y = 0$$

La solució consisteix en incloure els infinits dins l'aritmètica, adoptant diverses regles senzilles tals com que 1/0=Inf, 1/Inf=0, $x$+Inf=Inf, etc. D'aquesta manera, la seqüència d'operacions de l'exemple anterior serà: $100/x$ = Inf, 1+Inf = Inf, i 1/Inf = 0.

Per convenció, l'infinit es representa amb tots els bits de la fracció a zero i els de l'exponent a 1:

F = 000 . . . 0, E = 111 . . . 1

**c) NaN**

Les operacions amb nombres reals no sempre produeixen resultats vàlids. Cal no confondre aquests resultats invàlids amb els overflows o underflows. Ens referim a operacions amb resultats indeterminats perquè els operands no pertanyen al seu domini (per exemple l'arrel quadrada d'un nombre negatiu, el logaritme d'un nombre negatiu o zero, el quocient 0/0), i també algunes operacions amb infinits com Inf-Inf, 0×Inf, Inf/Inf.

Per evitar que una operació invàlida causi una excepció i avorti l'execució del programa, es fa que el resultat adopti una codificació especial anomenada NaN (Not a Number) i que l'execució prossegueixi. A posteriori, el programa pot fer un test del resultat i prendre les decisions oportunes.

Per altra banda, en una cadena de càlculs resulta convenient per raons d'eficiència no haver de fer el test per a cada un dels resultats intermedis sinó que és molt millor fer el test només un cop, al resultat final. Per aquesta raó l'estàndard defineix una sèrie de regles de propagació dels NaN al llarg de les cadenes d'operacions. En general, qualsevol operació en què un dels operands és un NaN, té com a resultat un altre NaN.

Per convenció, un NaN es representa amb tots els bits de l'exponent a 1, i amb una mantissa en què almenys un dels bits és no-nul (per diferenciar-se dels infinits).

F = xxx . . . x (no tots són zeros), E = 111 . . . 1

El valor concret que ha de tenir la mantissa no està especificat en l'estàndard[2]. En cas de propagar el NaN en una operació, tan sols es recomana que la mantissa del resultat sigui igual a la d'un dels operands.

**d) Denormals**

A la secció 4.5 hem vist que l'error relatiu de precisió dels nombres en coma flotant està afitat i que aquesta fita depèn exclusivament de la precisió de la mantissa. Però hem vist també que en cas d'underflow, aquesta propietat no es compleix i l'error pot arribar a ser 6 ordres de magnitud major, fent perillar la fiabilitat del resultat. La solució a aquest problema consisteix a admetre la representació d'un conjunt de nombres en l'interval $(0, 2^{Emin})$ que "omplin" aquest gran forat. Això s'aconsegueix admetent nombres no-normalitzats o *denormals* en aquest interval, amb valors absoluts de la forma

$$v = 0{,}xxx \ldots x \times 2^{Emin}$$

Aquests nombres ocupen tot l'interval $(0, 2^{Emin})$ de forma uniforme. El màxim error absolut és menor que la diferència entre valors representables consecutius

$$\varepsilon_{max} < 0{,}0 \ldots 001 \times 2^{Emin} = 2^{Emin-23}$$

Notem que aquesta fita d'error absolut és exactament igual a la que es produeix en l'interval $(2^{Emin}, 2^{Emin+1})$.

2. La revisió de 2008 de l'estàndard estableix que el primer bit de la fracció sigui 1 per a un "quiet NaN", o bé 0 per a un "signaling NaN". En aquest curs no estudiarem aquesta distinció.

Si *v* pertany a qualsevol interval de nombres representables consecutius ($v_0$, $v_1$) tals que $v_0 \neq 0$, la fita d'error relatiu és

$$\eta_{max} < \varepsilon_{max} / |v_0| < 2^{Emin-23} / |v_0|$$

En canvi, dins l'interval $(0, 2^{Emin-23})$ l'error relatiu $\eta = \varepsilon / |v|$ no està afitat ja que el valor exacte *v* pot ser un nombre infinitament petit. No obstant, sense els nombres denormals, si disminuïm la magnitud *v* per sota del llindar de l'underflow ($2^{Emin}$), la fita d'error absolut augmenta abruptament, i també ho fa l'error relatiu. En canvi, adoptant els nombres denormals, la fita d'error absolut dins l'interval $(0, 2^{Emin})$ es manté constant i l'error relatiu augmenta progressivament a mesura que *v* s'acosta a zero. És per aquesta raó que aquest mecanisme es coneix també com "underflow gradual". Així per exemple, en simple precisió, la fita d'error relatiu és $2^{-23}$ a l'extrem superior de l'interval i va creixent a $2^{-22}$, $2^{-21}$, $2^{-20}$... $2^{0}$ a mesura que disminueix l'ordre de magnitud de *v* (i a mesura que disminueix també el nombre de dígits significatius de la mantissa no-normalitzada).

Els nombres denormals es representen amb l'exponent que té tots els bits a zero, i una mantissa en què al menys un dels bits és no-nul (per diferenciar-se del zero).

F = xxx . . . x (no tots són zeros), E = 000 . . . 0

A l'hora de fer operacions amb un nombre denormal, cal tenir en compte que la matissa no té bit ocult sinó que val 0,F i que l'exponent implícit és $E_{min}$. El següent quadre resumeix els formats dels quatre casos especials i el cas normal:

| | | **Exponent (E)** | | |
|---|---|---|---|---|
| | | Tot 0 | Altres | Tot 1 |
| **Mantissa (F)** | Tot 0 | Zero | Normalitzat | Infinit |
| | Altres | Denormal | | NaN |

## 4.7 Arrodoniment

Com ja hem vist, el resultat exacte *v* d'una operació pot no ser representable amb un nombre finit de bits de la mantissa, raó per la qual haurem d'optar per aproximar-lo per un dels dos valors representables més pròxims $v_0$ o $v_1$, essent $v_0 < v < v_1$. Aquesta aproximació rep el nom d'arrodoniment. L'estàndard defineix 4 modes d'arrodoniment per triar entre $v_0$ o $v_1$: el valor que sigui més pròxim a *v*, a zero, a +Infinit, o a -Infinit. Per defecte, el hardware està configurat sempre per usar l'arrodoniment al més pròxim.

### a) Arrodoniment al més pròxim (per defecte)

Aquest mode proporciona el menor error de precisió possible, el qual serà menor que la meitat de la llargada de l'interval ($v_0$, $v_1$), és a dir

$$\varepsilon_{max} < |v_1 - v_0| / 2$$

I, seguint el que hem vist a la secció 4.5 l'error relatiu serà

$$\eta_{max} < 0{,}5 \text{ ULP}$$

Per al format de simple precisió és $\eta_{max} < 2^{-24}$, i per al de doble precisió $\eta_{max} < 2^{-53}$

Suposem que en base 10 tenim el nombre $v$ = 13,375. Si l'hem de representar amb 2 dígits significatius l'arrodonim a 13; amb 3 dígits l'arrodonim a 13,4 (ja que s'aproxima més que no 13,3); i si el volem amb 4 dígits el podem arrodonir a 13,38 o bé a 13,37, ja que els dos valors són exactament equidistants (encara que en molts contextos aquest cas és costum arrodonir-lo a 13.38).

Vegem els següents exemples de resultats exactes amb 6 bits significatius i que cal arrodonir a 3 bits, triant entre 101 o bé 110 (en decimal, entre 5 o 6).

```
101,010  ->  101    (en decimal: 5,250  -> 5)
101,011  ->  101    (en decimal: 5,375  -> 5)
```

Els hem arrodonit cap al valor anterior per truncament. Sabem que és el més pròxim perquè el primer bit eliminat (a la dreta de la coma) és un 0.

```
101,101  ->  110    (en decimal: 5,625  -> 6)
```

L'hem arrodonit cap al valor següent sumant-li 1. Sabem que és el més pròxim perquè el primer bit eliminat (a la dreta de la coma) és un 1, i més a la dreta encara hi almenys un altre 1 (això fa que no sigui exactament equidistant).

```
101,100  ->  ?      (en decimal: 5,500  -> ?)
```

Aquest cas és equidistant de l'anterior i del següent. Ho sabem perquè el primer bit eliminat és un 1 i la resta són zeros. Vist individualment, sembla que podríem optar per qualsevol dels dos, si triem 110 l'error serà positiu i si triem 101 serà negatiu. Però estadísticament, per a una cadena de càlculs estaríem introduint un biaix positiu o negatiu si sempre decidim en el mateix sentit. Seria desitjable que en la meitat de casos optéssim per l'error positiu i en l'altra meitat pel negatiu, i així aconseguir que es cancel·lessin. Una solució òbvia seria decidir-ho a l'atzar, que per definició dóna igual probabilitat a cada opció. Però aquest criteri introduiria una indeterminació en el resultat dels càlculs: el mateix càlcul repetit en diverses ocasions podria donar resultats diferents, i en dificultaria la verificació. En comptes d'això, l'estàndard va optar per un altre criteri que dóna la mateixa probabilitat a cada opció però és determinista: triar aquell dels dos valors que sigui parell (acabat en zero). Així doncs, en el nostre exemple

```
101,100  ->  110      (en decimal: 5,500  -> 6)
```

L'hem arrodonit cap al valor següent sumant-li 1, perquè aquest és el valor parell.

Vegem exemples anàlegs amb mantisses de 27 bits que cal arrodonir a 24:

```
1,xxx . . . 101010 -> 1,xxx . . . 101  (a l'anterior)
1,xxx . . . 101011 -> 1,xxx . . . 101  (a l'anterior)
1,xxx . . . 101100 -> 1,xxx . . . 110  (equidistant: al parell)
1,xxx . . . 101101 -> 1,xxx . . . 110  (al següent)
```

### b) Altres modes d'arrodoniment

El segon mode aproxima el resultat al valor de menor magnitud, és a dir aquell que estigui més pròxim al zero. L'error relatiu és $\eta_{max} < 1$ ULP, pitjor que amb l'arrodoniment al més pròxim, però és molt més simple d'implementar: truncar els bits que sobren.

El tercer mode aproxima el resultat al valor major, és a dir al més pròxim a +Infinit. I el quart mode aproxima al valor menor, és a dir al més pròxim a -Infinit. Aquests dos modes en conjunt s'usen en *aritmètica d'intervals*, un mètode de la matemàtica computacional que posa límits als errors d'arrodoniment per tal d'assolir resultats fiables. Per resumir-ho, aquest mètode representa cada valor per un interval de possibilitats.

## 4.8 Exemple de conversió de base 10 a coma flotant

Es demana codificar en coma flotant de simple precisió el número $v = -1029{,}68$ i determinar l'error de precisió comès en la conversió.

### a) Convertir a binari la part entera

Per divisions successives obtenim: $1029 = 10000000101_2$ (consta de 11 bits)

### b) Convertir a binari la part decimal

Apliquem l'algorisme de multiplicacions successives per 2. Comencem multiplicant $0{,}68 \times 2 = \mathbf{1}{,}36$: el primer bit fraccionari és 1. L'algorisme continua, multiplicant $0{,}36 \times 2$, etc. Els successius resultats de l'algorisme són (en negreta el bit obtingut en cada pas):

**1**,36; **0**,72; **1**,44; **0**,88; **1**,76; **1**,52; **1**,04; **0**,08;

**0**,16; **0**,32; **0**,64; **1**,28; **0**,56; **1**,12; **0**,24; **0**,48;

**0**,96; **1**,92 . . .

I encara podríem seguir indefinidament. El resultat que tenim de moment és

$v = -10000000101{,}101011100001010001\ldots$

### c) Normalitzar la mantissa

$v = -1{,}00000001011010111000010\mathbf{10001}\ldots \times 2^{10}$

### d) Arrodonir la mantissa

En simple precisió la mantissa sols pot tenir 24 bits, però aquest resultat (ja en té 29 i en podria tenir més) no es pot representar de forma exacta amb 24 bits i cal arrodonir-lo. Per defecte, l'estàndard estipula arrodonir al més pròxim i, com hem vist en la secció anterior, això requereix considerar un nombre suficient de bits extra que permeti determinar si el valor representable més pròxim és el l'anterior o el següent. En el nostre exemple hem aturat les multiplicacions després d'obtenir els primers 29 bits perquè dels bits a descartar 10001 (en negreta a dalt) ja podem deduir que cal arrodonir al valor següent (però amb 28 bits no hauríem pogut!). Aquest s'obté doncs sumant-li 1 ULP al valor truncat:

```
  1,00000001011010111000010
+                         1
---------------------------
  1,00000001011010111000011
```

### e) Codificar l'exponent

Calculem el valor explícit de l'enter E=10, en excés a 127, i codifiquem-lo en binari

$E_u = 10+127 = 137 = 10001001_2$

### f) Ajuntem Signe, Exponent i Fracció (però descartant el bit ocult!)

$V = 1\ 10001001\ 00000001011010111000011_2$

Expressat en hexadecimal queda

$V = 1100\ 0100\ 1000\ 0000\ 1011\ 0101\ 1100\ 0011_2 = 0xC480B5C3$

### g) Calcular l'error de precisió

L'error de precisió és la diferència en valor absolut entre el valor exacte i el representat. Així doncs,

$$\begin{aligned}\varepsilon &= (\ 1,00000001011010111000011 \\ &\quad - 1,00000001011010111000010\mathbf{10001}\ ) \times 2^{10} \\ &= \ 0,00000000000000000000001111\ \times 2^{10} \\ &= \ 1,111 \times 2^{-15}\end{aligned}$$

Expressat en decimal,

$$\varepsilon = 1,875 \times 2^{-15} = 1,875/32768 = 5,722 \times 10^{-5}$$

## 4.9 Exemple de conversió de coma flotant a base 10

Es demana convertir a decimal el número en coma flotant de simple precisió $V = \texttt{0x45814140}$. Per començar, l'escrivim en binari,

$$V = 0100\ 0101\ 1000\ 0001\ 0100\ 0001\ 0100\ 0000_2$$

### a) Decodificar signe, mantissa i exponent

Agrupem els camps de bits de Signe, Exponent i Fracció,

```
V = 0 10001011 00000101010000010100000
    - -------- -----------------------
    S    E              F
```

El signe és S = positiu. El valor explícit de l'exponent (natural) és $E_u = 10001011_2 = 139$. Per tant, l'exponent (enter) és E = 139 - 127 = 12. Afegint el bit ocult a la fracció, obtenim la mantissa: $1,00000010100000101000000$. Així doncs, el número és

$$V = +1,00000010100000101000000_2 \times 2^{12}$$

### b) Eliminar la potència

Efectuem el producte movent la coma 12 posicions a la dreta,

$$V = 1000000101000,00101_2$$

### c) Convertir la part entera a decimal

Convertim la part entera a decimal fent la suma ponderada dels bits

$$1000000101000_2 = 1 \cdot 2^{12} + 1 \cdot 2^5 + 1 \cdot 2^3 = 4096 + 32 + 8 = 4136$$

### d) Convertir la part fraccionària a decimal

Convertim la part fraccionària en entera movent la coma a la dreta. Convertim el coeficient a decimal, i efectuem el producte per la potència de 2 (o divisió en aquest cas)

$$0,00101_2 = 101_2 \times 2^{-5} = 5/32 = 0,15625$$

Ajuntant les dues parts, resulta el número decimal

$$v = 4136,15625$$

3.5

# 5. Suma o resta en coma flotant

L'algorisme de suma i resta segueix les mateixes regles que el que estem acostumats a aplicar en base 10.

## 5.1 Suma en coma flotant, en base 10

Vegem el següent exemple en base 10, amb 4 dígits de precisió a la mantissa. Volem sumar $z = x + y$, essent $x = 9{,}999 \times 10^{1}$, $y = 1{,}680 \times 10^{-1}$

Per sumar-los cal que els exponents siguin iguals. En realitat podem triar qualsevol exponent com a exponent comú als dos números, però l'estàndard estipula igualar-los al major dels dos (a 1). Així que el segon sumand s'ha de convertir

$$y = 1{,}680 \times 10^{-1} = 0{,}01680 \times 10^{1}$$

Sumem les mantisses:

$$\begin{array}{rl} & 9{,}999 \times 10^{1} \\ + & 0{,}01680 \times 10^{1} \\ \hline z = & 10{,}01580 \times 10^{1} \end{array}$$

Normalitzem el resultat i comprovem un possible overflow/underflow

$$z = 1{,}001580 \times 10^{2}$$ (exponent dins el rang, no hi ha excepció)

Finalment arrodonim el resultat a la precisió convinguda (4 dígits de mantissa). Sobren 3 bits (en negreta), i seguim el mètode d'arrodonir al més pròxim (en aquest cas, al següent)

$$z = 1{,}001\mathbf{580} \times 10^{2} = 1{,}002 \times 10^{2}$$

## 5.2 Suma en el format de simple precisió

Es demana sumar $z = x + y$, essent $x =$ 0x3F40000D, $y =$ 0xC0800004. Per començar, els expressem en base 2:

```
x = 0011 1111 0100 0000 0000 0000 0000 1101
y = 1100 0000 1000 0000 0000 0000 0000 0100
```

Separant els camps de signe, exponent i fracció

```
x = 0 01111110 10000000000000000001101
y = 1 10000001 00000000000000000000100
```

Decodifiquem els exponents. L'exponent de $x$ té el valor explícit (natural) $E_u = 01111110_2 = 126$. Per tant el seu valor (enter) és E = 126-127 = -1. L'exponent de $y$ té el valor explícit (natural) $E_u = 10000001_2 = 129$. Per tant el seu valor (enter) és E = 129-127 = 2.

Afegint els signes i els bits ocults de les mantisses (en negreta) podem expressar els dos números com

$$x = \mathbf{+1}{,}10000000000000000001101 \times 2^{-1}$$
$$y = \mathbf{-1}{,}00000000000000000000100 \times 2^{2}$$

### a) Igualar exponents (al major)

El major exponent és el de y, i val 2. Per igualar l'exponent de *x* a 2 caldrà moure la coma 3 llocs a l'esquerra:

$$x = +1,10000000000000000001101 \times 2^{-1}$$
$$= +0,00110000000000000000001101 \times 2^{2}$$

### b) Sumar o restar les magnituds

Segons siguin els signes iguals o diferents l'operació a fer serà una suma o una resta. En el nostre exemple caldrà restar. A més a més, en cas de ser de diferents signes, hem de restar el de major magnitud menys el de menor magnitud, independentment de quin d'ells sigui el negatiu, i al final caldrà assignar al resultat el signe de l'operand de major magnitud. En el nostre cas el de major magnitud és y. Així doncs restem

$$\begin{array}{rl} & |y| = 1,00000000000000000000100 \times 2^{2} \\ - & |x| = 0,00110000000000000000001\mathbf{101} \times 2^{2} \\ & \text{----------} \\ = & |z| = 0,11010000000000000000010\mathbf{011} \times 2^{2} \end{array}$$

### c) Normalitzar la mantissa

Si el resultat no ho està, caldrà normalitzar-lo, movent la coma en un sentit o en l'altre i ajustant l'exponent

$$|z| = 1,10100000000000000000100\mathbf{11} \times 2^{1}$$

Com a conseqüència d'aquesta operació pot resultar un underflow (si $E < E_{min}$) o un overflow (si $E > E_{max}$), causant la corresponent excepció.

### d) Arrodonir la mantissa

A fi de donar el resultat el més exacte possible se solen fer els càlculs intermedis amb més bits dels que té el format estàndard. Aquests bits reben el nom de *bits de guarda* (en negreta, en l'exemple anterior). Si en acabar el càlcul els bits de guarda no són tots zeros, vol dir que el resultat exacte no es pot representar amb la precisió demanada i cal arrodonir-lo. En el nostre exemple, els bits de guarda són `11`, i arrodonim cap al valor representable següent, sumant 1 ULP al valor truncat:

$$\begin{array}{rl} & 1,10100000000000000000100 \times 2^{1} \\ + & \qquad\qquad\qquad\qquad\qquad 1 \\ & \text{----------} \\ |z| = & 1,10100000000000000000101 \times 2^{1} \end{array}$$

L'arrodoniment pot causar que el resultat no estigui normalitzat, i en aquest cas cal tornar-lo a normalitzar i arrodonir.

### e) Codificar signe, exponent i mantissa

El valor explícit de l'exponent E=1 és $E_u = 1+127 = 128 = 10000000_2$. El signe del resultat és el de l'operand de major valor absolut, que és *y* i és negatiu. Queda doncs:

$$Z = 1\ 10000000\ 10100000000000000000101$$
$$= 1100\ 0000\ 0101\ 0000\ 0000\ 0000\ 0000\ 0101$$
$$= 0xC0500005$$

### f) Calcular l'error de precisió de la suma

L'error de precisió és la diferència en valor absolut entre el valor exacte i el representat després de l'arrodoniment. Així doncs,

$$\begin{aligned}\varepsilon &= (\ 1{,}10100000000000000000101 \\ &\quad - 1{,}10100000000000000000100\mathbf{11}\ ) \times 2^{1} \\ &= \quad 0{,}00000000000000000000000001\ \times 2^{1} \\ &= \quad 0{,}00000000000000000000000001 \\ &= \quad 1{,}0 \times 2^{-24}\end{aligned}$$

Expressat en decimal,

$$\varepsilon = 6{,}0 \times 10^{-8}$$

## 5.3 Bits de guarda

Durant les operacions de suma i resta, si els operands tenen diferents exponents, es desplaça la mantissa del de menor magnitud cap a la dreta un cert número de posicions. Si no volem descartar cap bit a fi de minimitzar l'error de precisió, el sumador/restador haurà de preveure tants half-adders addicionals com posicions resulti desplaçada la mantissa. Els bits addicionals als propis de la mantissa s'anomenen *bits de guarda*. En el cas pitjor (dos operands amb exponents tan diferents com $E_{max}$ i $E_{min}$), faria falta un sumador/restador amb un nombre exagerat de bits (277, en simple precisió!).

Es podria reduir el cost (en temps i en circuits) descartant tots els bits addicionals que apareguin a la dreta a conseqüència del desplaçament de la mantissa abans d'operar. En el cas de la suma, és fàcil veure que descartant els bits sempre obtindrem el mateix resultat que si operem amb tots els bits i arrodonim el resultat: els bits apareguts a la dreta de l'operand desplaçat no generen cap *carry* ja que se sumen a bits a zero de l'altre operand. En canvi, la resta sí que pot produir resultats erronis si descartem els bits:

**Sense descartar cap bit**

```
  1,10001
- 0,00100 101
  1,01100 011
```

**Arrodonir:**

$\boxed{1{,}01100}$

**Descartant els bits "sobrants"**

```
  1,10001
- 0,00100 101
  1,01101
```

**Incorrecte!**

Però ¿com d'important pot arribar a ser l'error introduït si descartem els bits? Un cas extrem el trobem en el següent exemple, que resta $1{,}00000 \times 2^{0} - 1{,}11111 \times 2^{-1}$:

**Sense descartar cap bit**

```
  1,00000
- 0,11111 1
  0,00000 1
```

**Normalitzar:**

$\boxed{1{,}0 \times 2^{-6}}$

**Descartant els bits "sobrants"**

```
  1,00000
- 0,11111 1
  0,00001
```

**Normalitzar:**

$\boxed{1{,}0 \times 2^{-5}}$

L'error absolut és $\varepsilon = 2^{-5} - 2^{-6} = 2^{-6}$, i l'error relatiu és $\eta = \varepsilon / 2^{-6} = 1$. L'error és tan gran com el resultat! És inacceptable, però tampoc volem un sumador/restador gegant.

Observem que sovint, després d'arrodonir el resultat d'una resta calculat amb tots els bits, la major part són descartats sense afectar al resultat final. Resulta per tant lògic preguntar-se si és possible calcular el mateix resultat arrodonit que amb el restador gegant, però amb el mínim de bits de guarda.

Com es demostra fàcilment en l'apèndix A d'aquest tema, en les operacions de resta sols cal operar 3 bits de guarda, que anomenem Guard (G), Round (R) i Sticky (S). Els bits G i R són els dos primers bits de guarda del substraend. El bit S es forma substituint el tercer bit de guarda per la "or" lògica d'aquest bit i tots els que té a la seva dreta.

Així doncs, podem resumir l'algorisme de suma/resta amb tres bits de guarda així: per a un format de coma flotant amb *p* dígits de mantissa (incloent el bit ocult) cal un sumador/restador de *p*+3 bits. Abans d'iniciar l'operació, extenem la precisió de les dues mantisses amb 3 zeros a la dreta (bits Guard, Round i Sticky). A continuació restem els dos exponents per determinar quin és l'operand de menor magnitud –que serà el substraend en cas de resta– i quantes posicions a la dreta s'ha de desplaçar. Després del desplaçament, el bit Sticky (el bit de menor pes de la mantissa de *p*+3 bits) ha de contenir la 'or' lògica del bit en aquesta posició i tots els bits a la seva dreta que hauran estat eliminats pel desplaçament. Els restants passos de l'algorisme es realitzaran com si la mantissa tingués *p*+3 bits fins a l'arrodoniment, que la reduirà a *p* bits. Aquest procediment garanteix que l'error de precisió del resultat és el mateix que operant amb tots els bits. Vegem un exemple amb precisió de 6 bits:

**Restar amb tots els bits**

```
   1,00000    Sticky=1
 - 0,00001 10011
   0,11110 01101
```

**Normalitzar:**

```
   1,11100 1101
```

**Arrodonir (al següent):**

```
  1,11101
```

**Restar amb bits Guard, Round i Sticky**

```
             GRS
   1,00000
 - 0,00001 101
   0,11110 011
```

**Normalitzar:**

```
   1,11100 11
```

**Arrodonir (al següent):**

```
  1,11101
```

3.5

# 6. Multiplicació o divisió en coma flotant

L'algorisme de multiplicació o divisió segueix les mateixes regles que el que estem acostumats a aplicar en base 10.

## 6.1 Multiplicació (o divisió) en coma flotant, en base 10

Vegem el següent exemple en base 10, amb 4 dígits de precisió a la mantissa i 2 dígits d'exponent. Volem multiplicar $z = x \times y$, essent $x = -2{,}111 \times 10^{10}$, $y = 9{,}200 \times 10^{-5}$

$$z = -2{,}111 \times 10^{10} \cdot 9{,}200 \times 10^{-5} =$$
$$= -(2{,}111 \cdot 9{,}200) \times 10^{(10-5)}$$

Per a la multiplicació (o divisió) en coma flotant s'han de sumar (o restar) els exponents, multiplicar (o dividir) les mantisses, i assignar signe negatiu al resultat en cas que els números siguin de diferent signe. No oblidem també normalitzar i arrodonir el resultat.

Sumem els exponents: $10-5=5$

Multipliquem les mantisses:

$$\begin{array}{r} 2111 \\ \times\ 9200 \\ \hline 0000 \\ 0000 \\ 4222 \\ 18999 \\ \hline 19421200 \end{array}$$

Resultat, amb la coma: $19{,}421200 \times 10^{5}$

Normalitzem el número i comprovem un possible overflow/underflow:

$|z| = 1{,}9421200 \times 10^{6}$ (exponent dins el rang [-99,+99])

L'arrodonim a 4 dígits de precisió (els bits que sobren, en negreta) a l'anterior:

$|z| = 1{,}942\mathbf{1200} \times 10^{6} \cong 1{,}942 \times 10^{6}$

I li afegim el signe:

$z = -1{,}942 \times 10^{6}$

## 6.2 Multiplicació (o divisió) en el format de simple precisió

Es demana multiplicar $z = x \times y$, essent $x =$ 0x3F600000, $y =$ 0xBED00002. Per començar, els expressem en base 2:

```
x = 0011 1111 0110 0000 0000 0000 0000 0000
y = 1011 1110 1101 0000 0000 0000 0000 0010
```

Separant els camps de signe, exponent i fracció

```
x = 0 01111110 11000000000000000000000
y = 1 01111101 10100000000000000000010
```

Decodifiquem els exponents. L'exponent de $x$ té el valor explícit (natural) $E_u = 01111110_2 = 126$. Per tant el seu valor (enter) és E = 126-127 = -1. L'exponent de $y$ té

el valor explícit (natural) $E_u = 01111101_2 = 125$. Per tant el seu valor (enter) és E = 125-127 = -2.

Afegint els signes i els bits ocults de les mantisses (en negreta) podem expressar els dos números com

$x = \mathbf{+1},11000000000000000000000 \times 2^{-1}$
$y = \mathbf{-1},10100000000000000000010 \times 2^{-2}$

### a) Sumar els exponents

$E = -1+(-2) = -3$

### b) Multiplicar les mantisses

Per claredat, hem eliminat els zeros finals del primer operand

```
      1110
×     1101000000000000000000010
                           1110
      1110
    1110
  1110
10110110000000000000000011110
```

Resultat, amb la coma: $10,11011000000000000000011110 \times 2^{-3}$

### c) Normalitzar

$|z| = 1,01101100000000000000011110 \times 2^{-2}$

I comprovem un possible overflow/underflow: $-2 \in [-126,+127]$

### d) Arrodonir la mantissa

Arrodonim a 24 dígits de precisió (els bits que sobren, en negreta) al següent

```
      1,01101100000000000000001 1110 × 2^-2
   +                            1
    -------------------------
|z| ≅  1,01101100000000000000 10     × 2^-2
```

### e) Afegir el signe:

$z = -1,01101100000000000000010 \times 2^{-2}$

### f) Codificar signe, exponent i mantissa

El valor explícit de l'exponent E=-2 és $E_u = -2+127 = 125 = 01111101_2$. Ajuntant signe, exponent i mantissa en el format de simple precisió queda

$z$ = 1 01111101 01101100000000000000010
= 1011 1110 1011 0110 0000 0000 0000 0010
= 0xBEB60002

3.5

# 7. La coma flotant en MIPS

En els primers processadors, degut a l'escassetat de transistors dels circuits integrats, les unitats funcionals de coma flotant no s'integraven en el xip de la CPU sinó que formaven una unitat específica ubicada en un xip opcional a part. La unitat de coma flotant funcionava com un coprocessador, amb el seu propi banc de registres i unitats funcionals. Aquesta disposició perviu encara en l'ISA de MIPS a pesar que avui en dia aquests circuits ja s'han integrat en el xip de la CPU.

Des del punt de vista de l'ISA doncs, la unitat de coma flotant del MIPS és un coprocessador (CP1) que disposa del seu propi banc de registres de 32 bits ($f0 - $f31) i les unitats funcionals necessàries per executar un conjunt específic d'instruccions que operen amb números en coma flotant de simple precisió (dades de tipus *float*, en C) o doble precisió (dades de tipus *double*, en C). Les instruccions que operen amb números de doble precisió especifiquen només registres parells, sobreentenent que el número a operar està repartit entre l'esmentat registre (bits de menor pes) i el següent (bits de major pes)[3].

A part dels registres de propòsit general, el CP1 conté un Registre de Control amb diversos camps de bits. Hi uns bits on queden registrades les excepcions ocorregudes en la darrera instrucció executada, a fi que puguin ser consultats per instruccions posteriors. Uns altres bits permeten decidir per cada tipus d'excepció (overflow, underflow, resultat invàlid, etc) si ha d'interrompre l'execució i ser tractada amb una rutina específica o bé simplement produir un resultat convingut per defecte (Infinit, Zero, NaN, etc). Un tercer camp de bits permet establir el mode d'arrodoniment. I finalment, hi ha un bit anomenat Bit de Condició (bit CC) que guarda el resultat de les instruccions de comparació. Aquest bit és l'operand destinació implícit en aquest tipus d'instruccions.

## 7.1 Repertori d'instruccions de coma flotant

### a) Còpia entre registres

| | | |
|---|---|---|
| `mfc1 rt,fs` | rt = fs | còpia CPU ← CP1 |
| `mtc1 rt,fs` | fs = rt | còpia CP1 ← CPU |
| `mov.s fd, fs` | fd = fs | còpia CP1 ← CP1 |

### b) Accés a memòria

El registre base és un registre de propòsit general de la CPU

| `lwc1/ldc1/swc1/sdc1` | | |
|---|---|---|
| `lwc1 ft, off16(rs)` | ft = $M_w$[rs + SignExt(off16)] | load float |
| `ldc1 ft, off16(rs)` | ft = $M_d$[rs + SignExt(off16)] | load double |
| `swc1 ft, off16(rs)` | $M_w$[rs + SignExt(off16)] = ft | store float |
| `sdc1 ft, off16(rs)` | $M_d$[rs + SignExt(off16)] = ft | store double |

3. Aquesta restricció limita a 16 el nombre total de números de doble precisió que es poden guardar en registres. Posteriors ampliacions de l'ISA van suprimir aquesta restricció, permetent guardar un total de 32 números en doble precisió.

### c) Aritmètiques

| | | |
|---|---|---|
| `add.s fd, fs, ft` | fd = fs + ft | suma floats |
| `add.d fd, fs, ft` | fd = fs + ft | suma doubles |
| `sub.s fd, fs, ft` | fd = fs - ft | resta floats |
| `sub.d fd, fs, ft` | fd = fs - ft | resta doubles |
| `mul.s fd, fs, ft` | fd = fs × ft | multiplica floats |
| `mul.d fd, fs, ft` | fd = fs × ft | multiplica doubles |
| `div.s fd, fs, ft` | fd = fs / ft | divideix floats |
| `div.d fd, fs, ft` | fd = fs / ft | divideix doubles |

### d) Comparacions

L'operand destinació (implícit) d'aquestes instruccions és el Bit de Condició (bit CC del registre de control FCR).

| `c.eq.s/c.eq.d/c.lt.s/c.lt.d/c.le.s/c.le.d` | | |
|---|---|---|
| `c.eq.s fs,ft` | bit de condició cc=1 si fs==ft, sino cc=0 | igual que float |
| `c.eq.d fs,ft` | bit de condició cc=1 si fs==ft, sino cc=0 | igual que double |
| `c.lt.s fs,ft` | bit de condició cc=1 si fs < ft, sino cc=0 | menor que float |
| `c.lt.d fs,ft` | bit de condició cc=1 si fs < ft, sino cc=0 | menor que double |
| `c.le.s fs,ft` | bit de condició cc=1 si fs ≤ ft, sino cc=0 | menor o igual que float |
| `c.le.d fs,ft` | bit de condició cc=1 si fs ≤ ft, sino cc=0 | menor o igual que double |

### e) Salts condicionals

Salten o no en funció del contingut del Bit de Condició.

| `bc1t/bc1f` | | |
|---|---|---|
| `bc1t etiqueta` | Salta si cc=1 (true) | |
| `bc1f etiqueta` | Salta si cc=0 (false) | |

## 7.2 Traducció de programes amb dades en coma flotant

Declaracions de dades globals en C:

```
float var1, var2[2] = {1.0, 3.14};
double var3 = -37.55
```

En assemblador MIPS

```
        .data
var1:   .float 0.0
var2:   .float 1.0, 3.14
var3:   .double -37.55
```

Les directives `.float` i `.double` alineen automàticament les dades a adreces múltiples de 4 i de 8 respectivament. Els valors inicials s'han d'escriure en decimal, si es volen escriure en hexadecimal cal emprar les directives `.word` o `.dword`.

Com ja s'ha estudiat al Tema 3, segons l'ABI de MIPS, si una funció té un o dos paràmetres de tipus float, aquests es passen en els registres $f12 i $f14, i si el resultat és un float, es retorna en $f0. Per altra banda, els registres del $f20 al $f31 són considerats com a "registres segurs" que han de ser retornats en el mateix estat inicial que s'han rebut.

## 7.3 Exemple de traducció

Traduir la següent funció declarada en C

```
float func(float x)
{
       if (x < 1.0)
              return x*x;
       else
              return 2.0 - x;
}
```

Observem que en el repertori de coma flotant cap instrucció té operands immediats, de manera que les constants 1.0 i 2.0 del codi anterior caldrà introduir-les en un registre de coma flotant (p.ex. $f16). Una estratègia consisteix en codificar-la a mà (p.ex. 1.0 = 0x3F800000) i escriure el següent codi:

```
       li     $t0, 0x3F800000
       mtc1   $t0, $f16
```

Alternativament, per estalviar-nos la feina de codificar a mà la constant, podem declarar una falsa variable en memòria:

```
       .data
const1:       .float 1.0
```

i llavors el codi de la funció quedarà així:

```
func:
       la     $t0, const1
       lwc1   $f16, 0($t0)
       c.lt.s $f12, $f16          # cc = (x < 1.0)
       bc1f   sino                # salta si cc=0
       mul.s  $f0, $f12, $f12     # return x*x
       b      fisi
sino:
       add.s  $f16, $f16, $f16    # $f16 = 1.0 + 1.0 = 2.0
       sub.s  $f0, $f16, $f12     # return 2.0 - x
fisi:
       jr     $ra
```

3.6

# 8. Paral·lelisme i associativitat

Paral·lelitzar un programa implica dividir les tasques en parts independents que es poden executar simultàniament en diferents unitats. Una llarga cadena de sumes és un bon candidat a ser descompost en subcadenes de sumes. Suposem una suma de 100 elements, amb un bucle que va acumulant el resultat: `(((x0+x1)+x2)+ ...)+x99`. Ara suposem que la descomponem en dues parts, una que calcula `((x0+x1)+...)+x49` i l'altra que calcula `((x50+x51)+...)+x99` i que les sumem al final.

Està clar que la transformació és matemàticament correcta per la propietat associativa de la suma. Si els números a sumar són enters en Ca2 no hi ha dubte que obtindrem el mateix resultat. Però cal preguntar-se si això mateix és cert per a números en coma flotant. La resposta és que no, degut als arrodoniments soferts en els càlculs intermedis.

Vegem-ho amb un contraexemple on comprovarem que $x+(y+z) \neq (x+y)+z$. Suposem que $x = -1{,}1 \times 2^{127}$, $y = 1{,}1 \times 2^{127}$, $z = 1{,}0$

$$\begin{aligned} x+(y+z) &= -1{,}1\times2^{127} + (1{,}1\times2^{127} + 1{,}0) \\ &= -1{,}1\times2^{127} + 1{,}1\times2^{127} \\ &= 0{,}0 \end{aligned}$$

Els operands que hem sumat dins el parèntesi són de magnituds tan dispars que un d'ells és menor que la fita d'error de precisió, i té una contribució nul·la en la suma.

$$\begin{aligned} (x+y)+z &= (-1{,}1\times2^{127} + 1{,}1\times2^{127}) + 1{,}0 \\ &= 0{,}0 + 1{,}0 \\ &= 1{,}0 \end{aligned}$$

En conseqüència, convé saber que la suma en coma flotant no és associativa, per tenir-ho en compte quan es verifiquen els resultats d'un programa que s'ha paral·lelitzat, ja que pot donar diferents resultats segons el nombre de processadors en què s'ha dividit l'execució. L'estudi de la credibilitat dels resultats s'estudia en Anàlisi Numèrica i constitueix per si mateix una branca de la informàtica.

No sempre els errors són tan grans, poden ser simples diferències en els bits de menor pes del resultat, poc significatives. Però no obstant, fins i tot si les diferències són mínimes, hi ha construccions de programació "perilloses" que poden amplificar els errors de manera catastròfica. Per exemple,

```
if (suma == 1.0)
       codi
```

Si `suma` és el resultat d'una llarga cadena d'operacions executada en paral·lel, pot ser que el salt condicional salti o no, depenent de com s'hagi paral·lelitzat. Una manera més prudent de programar el codi anterior hauria estat calcular una fita màxima `epsilon` d'error de la suma i comprovar si el resultat està dins l'interval $1.0 \pm$`epsilon`, en comptes de comparar si és estrictament igual a 1.0.

# Apendix A. Bits de guarda

Aquest Apèndix no pertany estrictament al programa d'EC, però pot servir per convèncer-se que 3 bits de guarda són suficients per a la màxima precisió.

Les operacions de resta de números en coma flotant poden introduir errors de precisió deguts al desplaçament a la dreta de la mantissa del substraend, en cas que la resta no consideri els bits addicionals que apareixen a la dreta a causa del desplaçament. No obstant, aquest desplaçament pot produir un nombre molt gran de bits addicionals (anomenats bits de guarda), i considerar-los tots pot augmentar exageradament la grandària i complexitat del restador. No obstant, veurem a continuació que tan sols són necessaris tres bits (Guard, Round i Sticky) per a fer la resta amb el mateix error de precisió que hauríem obtingut amb tots els bits.

### a) Borrow de la resta: bit Sticky

En la suma, els bits de guarda no produeixen mai "carry" i per tant no afecten a la mantissa final. En canvi, en la resta, el substraend pot quedar desplaçat a la dreta i els seus bits de guarda produiran un *borrow* que afectarà a la mantissa final (a menys que tots ells siguin zeros). Per tal de tenir en compte l'efecte d'aquest *borrow* sobre la mantissa final no és necessari fer la resta amb tots els bits, tan sols saber si algun dels bits de guarda és diferent de zero. Per tant, n'hi ha prou de restar amb 1 sol bit de guarda que representi la "or" lògica de tots els bits de guarda del substraend. Aquest bit rep el nom de "Sticky". En aquest exemple i en els següents hem suposat mantisses de $p$=6 bits de precisió.

**Resta amb tots els bits de guarda**

```
 1,00010
- 0,00000 0100011
 1,00001 1011101
```

**Resta sols amb bit Sticky**

```
          s
 1,00010
- 0,00000 1
 1,00001 1
```

**mateix borrow que tots els bits de guarda**

### b) Normalització: bit Guard

Es necessita normalitzar un resultat si la part entera de la mantissa és nul·la o té més d'un dígit. Si té més d'un dígit, es normalitza desplaçant-la a la dreta i no apareix cap bit de guarda en la mantissa final. En canvi, si és nul·la, la mantissa es normalitza desplaçant-la a l'esquerra i això fa que algun bit de guarda en passi a formar part. Però com es pot veure en els següents exemples, fins i tot en el pitjor cas, tan sols el primer dels bits de guarda (en negreta) passarà a formar part de la mantissa final. Aquest bit rep el nom de Guard. Els següents exemples il·lustren tres casos, fent les operacions amb tots els bits:

**A.- No desplaçat**

```
  1,00001
- 1,00000
  0,00001
```

**Normalitzar**:

```
 1,00000
```

**No hi ha bits de guarda**

**B.- Desplaçat 1 lloc**

```
  1,00000
- 0,11111 1
  0,00000 1
```

**Normalitzar**:

```
 1,00000
```

**1 bit de guarda en la mantissa final**

**C.- Desplaçat ≥ 2 llocs**

```
  1,00000
- 0,00111 101
  0,11000 011
```

**Normalitzar**:

```
 1,10000 11
```

**1 bit de guarda en la mantissa final**

En el cas A els exponents dels operands són iguals, no es desplaça el substraend i no apareixen bits de guarda. En el cas B la diferència d'exponents és 1, el substraend es desplaça, apareix 1 bit de guarda, i la normalització l'introdueix a la mantissa. En el cas C, la diferència d'exponents és major que 1, apareixen diversos bits de guarda, però la normalització no introdueix més que 1 sol bit de guarda a la mantissa. És fàcil veure que això és així per a qualsevol cas.

Aparentment doncs, sols el primer bit de guarda (bit Guard) influeix en el resultat, però si fem la resta sols amb el bit Guard, el resultat és 1,10001 i no és correcte. Això és degut a que aquest bit resulta afectat pel possible *borrow* procedent dels bits a la seva dreta. Tal com hem vist a l'apartat anterior, per a tenir en compte el *borrow* tan sols cal afegir un bit de Sticky. Vegem-ho en el següent exemple, on repetim el cas C, executat tan sols amb el bit Guard (G), o bé executat amb Guard i Sticky (GS):

**Exemple C, sols amb bit Guard**

```
          G
  1,00000
- 0,00111 1
  0,11000 1
```

**Normalitzar:**

1,10001

**Incorrecte!**

**Exemple C, amb bits Guard i Sticky**

```
          GS
  1,00000
- 0,00111 11
  0,11000 01
```

**Normalitzar:**

1,10000

**Correcte!**

En resum, que per a la normalització calen dos bits de guarda: Guard i Sticky.

### c) Arrodoniment: bits Round i Sticky

Com ja hem vist, l'arrodoniment al més pròxim pot presentar tres casos: que el més pròxim sigui l'anterior, el posterior, o equidistants. Els dos primers casos s'identifiquen pel valor del primer bit descartat i per tant requereixen obtenir el resultat amb 1 bit exacte addicional, que s'anomena "Round". Per altra banda, quan el bit Round val 1, cal distingir si hem d'arrodonir al següent (si existeix algun bit no nul a la dreta del bit Round) o bé es tracta del cas equidistant (tots els bits a la dreta del bit Round són zeros). Per distingir els dos casos es necessita un bit de guarda addicional que representi la "or" lògica de tots els bits a la dreta del bit Round. I aquest bit ja l'hem definit en el primer apartat, és l'Sticky. Els següents exemples il·lustren els tres casos, fent les operacions amb tots els bits

**Arrodonim a l'anterior**

```
  1,00001
+ 0,00000 0110001
  0,00001 0110001
```

**Arrodonir:** 1,00001

**Arrodonim al següent**

```
  1,00001
+ 0,00000 100001
  0,00000 100001
          S=1
```

**Arrodonir:** 1,00010

**Arrodonim al parell**

```
  1,00001
+ 0,00010 10000
  1,00011 10000
          S=0
```

**Arrodonir:** 1,00100

Repetim a continuació els mateixos exemples usant solament els bits Round i Sticky

**R=0 → a l'anterior**

```
          RS
  1,00001
+ 0,00000 01
  0,00001 01
```

**Arrodonir:** 1,00001

**R=1, S=1 → al següent**

```
          RS
  1,00001
+ 0,00000 11
  0,00000 11
```

**Arrodonir:** 1,00010

**R=1, S=0 → al parell**

```
          RS
  1,00001
+ 0,00010 10
  1,00011 10
```

**Arrodonir:** 1,00100