Tema 5. Aritmètica d'enters i coma flotant

Joan Manuel Parcerisa
Facultat d’Informàtica de Barcelona
Universitat Politècnica de Catalunya

Aritmètica d'enters

- Suma i resta
- Multiplicació
- Divisió

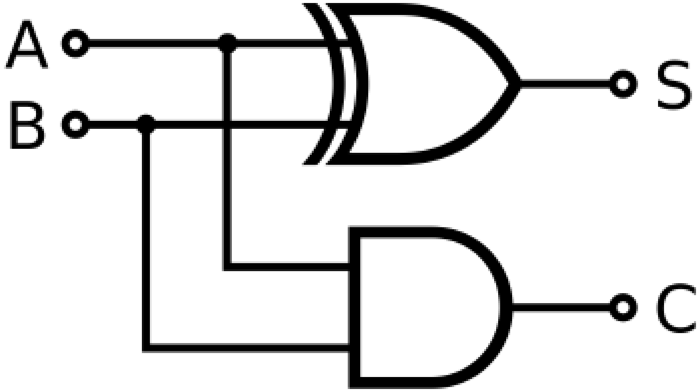
Suma de naturals i enters

A	B	Suma	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Suma de naturals i enters

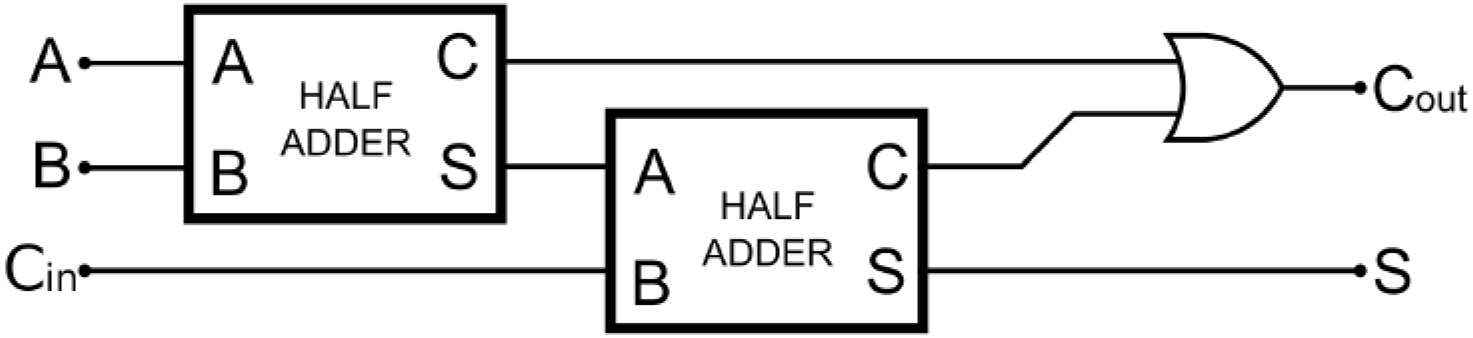
A	B	Suma	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- Half Adder



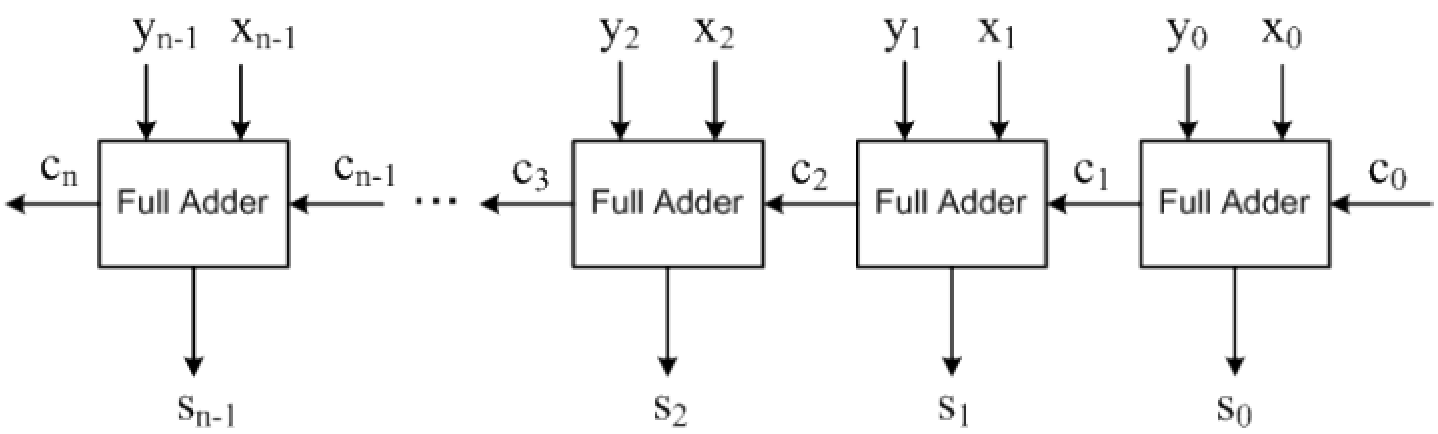
Suma de naturals i enters

- Full Adder



Suma de naturals i enters

- Sumador de *n* bits amb propagació de carry



- El mateix sumador per a naturals i enters en Ca2

Resta d'enters

- La resta

$$D = A - B$$

Equival a

$$D = A + (-B)$$

Resta d'enters

- La resta

  $D = A - B$

  Equival a

  $D = A + (-B)$

- Podem usar el sumador de n bits, canviant el signe de B
  - Invertir els bits de B (amb una porta NOT a cada bit)
  - Sumar 1 (fent el carry inicial $c_0=1$)

Overflow de suma i resta

- Rang d'enters en Ca2 amb *n* bits: $[-2^{n-1}, 2^{n-1}-1]$

Overflow de suma i resta

- Rang d'enters en Ca2 amb *n* bits: $[-2^{n-1}, 2^{n-1}-1]$
- Overflow d'una operació: quan el resultat no pertany al rang

  → El resultat amb n bits NO és correcte

Overflow de suma i resta

- Rang d'enters en Ca2 amb *n* bits: $[-2^{n-1}, 2^{n-1}-1]$
- Overflow d'una operació: quan el resultat no pertany al rang

  → El resultat amb n bits NO és correcte
- Overflow de la suma
  - Quan els operands són del mateix signe i el resultat és de signe contrari

Overflow de suma i resta

- Rang d'enters en Ca2 amb *n* bits: $[-2^{n-1}, 2^{n-1}-1]$
- Overflow d'una operació: quan el resultat no pertany al rang
  → El resultat amb n bits NO és correcte
- Overflow de la suma
  – Quan els operands són del mateix signe i el resultat és de signe contrari
- Resta:

$$diferència = minuend - substraend$$

$$minuend = substraend + diferència$$

Overflow de suma i resta

- Rang d'enters en Ca2 amb *n* bits: $[-2^{n-1}, 2^{n-1}-1]$
- Overflow d'una operació: quan el resultat no pertany al rang
  → El resultat amb n bits NO és correcte
- Overflow de la suma
  – Quan els operands són del mateix signe i el resultat és de signe contrari
- Resta:

$$diferència = minuend - substraend$$

$$minuend = substraend + diferència$$

- Overflow de la resta
  – Quan substraend i diferència són del mateix signe i el minuend és de signe contrari

Detecció d'overflow

- Per a naturals, overflow si c_n és igual a 1

$\oplus$

Detecció d'overflow

- Per a naturals, overflow si c_n és igual a 1
- Per a enters en Ca2, overflow si $c_{n-1} \neq c_n$

$$overflow = c_{n-1} \oplus c_n$$

Detecció d'overflow

- Per a naturals, overflow si c_n és igual a 1
- Per a enters en Ca2, overflow si $c_{n-1} \neq c_n$

$$overflow = c_{n-1} \oplus c_n$$

- Instruccions MIPS
  - `add`, `addi`, `sub` causen excepció en cas d'overflow d'enters
  - `addu`, `addiu`, `subu` ignoren els overflows

Detecció d'overflow

- Per a naturals, overflow si c_n és igual a 1
- Per a enters en Ca2, overflow si $c_{n-1} \neq c_n$

$$overflow = c_{n-1} \oplus c_n$$

- Instruccions MIPS
  - `add`, `addi`, `sub` causen excepció en cas d'overflow d'enters
  - `addu`, `addiu`, `subu` ignoren els overflows
- Alguns llenguatges, requereixen excepció en cas d'overflow d'enters
  - usaran `add`, `addi`, `sub` per a enters, i `addu`, `addiu`, `subu` per a naturals

Detecció d'overflow

- Per a naturals, overflow si c_n és igual a 1
- Per a enters en Ca2, overflow si $c_{n-1} \neq c_n$

$$overflow = c_{n-1} \oplus c_n$$

- Instruccions MIPS
  - `add`, `addi`, `sub` causen excepció en cas d'overflow d'enters
  - `addu`, `addiu`, `subu` ignoren els overflows
- Alguns llenguatges, requereixen excepció en cas d'overflow d'enters
  - usaran `add`, `addi`, `sub` per a enters, i `addu`, `addiu`, `subu` per a naturals

En C, s'ignoren els overflows
→ usarem `addu`, `addiu`, `subu` tant per a enters com naturals

Detecció d'overflow

- MIPS no inclou instruccions específiques per consultar si s'ha produït overflow

Detecció d'overflow

- MIPS no inclou instruccions específiques per consultar si s'ha produït overflow
- Es pot calcular per software: suposem la suma *s* = *a*+*b*

$$overflow = \overline{(a_{31} \oplus b_{31})} \cdot (a_{31} \oplus s_{31})$$

Detecció d'overflow

- MIPS no inclou instruccions específiques per consultar si s'ha produït overflow
- Es pot calcular per software: suposem la suma $s = a+b$

$$overflow = \overline{(a_{31} \oplus b_{31})} \cdot (a_{31} \oplus s_{31})$$

- En assemblador:

```
addu    $t2, $t0, $t1        # a + b
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         # mou el bit 31 a posició 0
```

Multiplicació de naturals

- En base 10

	348	multiplicand
x	951	multiplicador
	348	= 348 x 1
	1740	= 348 x 50
+	3132	= 348 x 900
	330948	

Multiplicació de naturals

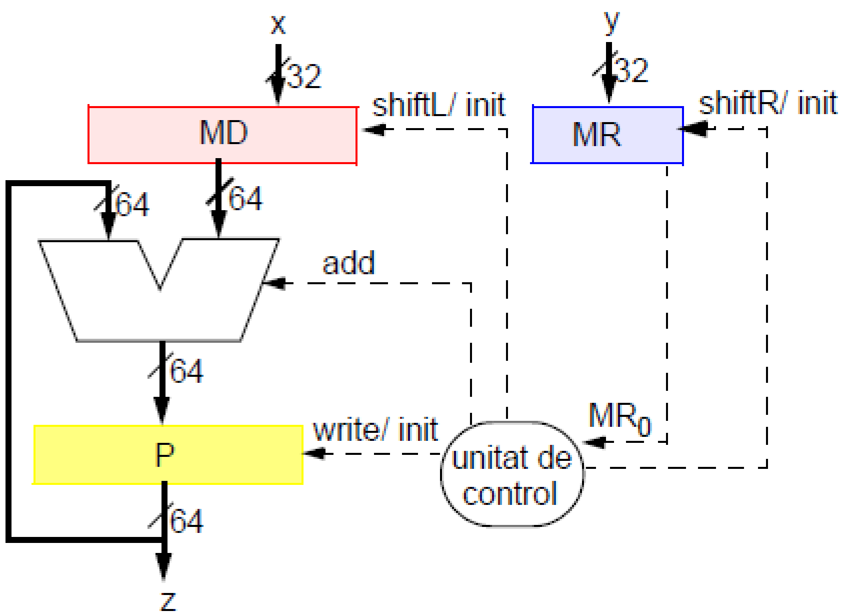
- En base 2

	1010	multiplicand
x	1101	multiplicador
	1010	= 1010 x 1
	0000	= 1010 x 00
	1010	= 1010 x 100
+	1010	= 1010 x 1000
	10000010	

Multiplicació de naturals: circuit

- Naturals de 32 bits amb resultat de 64 bits
  - Tarda 33 cicles... (suposant que una suma de 64 bits tarda 1 cicle)

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;
```

Multiplicació de naturals

- Exemple. Multipliquem x × y: 1010 × 1101

Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	**0**	**0**	**0**	**0**

Init: Inicialitzem $MD_{0:31}$ = x i MR = y
Desplacem MD a esquerra i MR a dreta

Multiplicació de naturals

- Exemple. Multipliquem x × y: 1010 × 1101

Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	**0**	**0**	**0**	**0**
1												**+**	0	0	0	0	**1**	**0**	**1**	**0**

Iter. 1: Com que **MR_0=1**, sumem P = P + MD
Desplacem MD a esquerra i MR a dreta

Multiplicació de naturals

- Exemple. Multipliquem x × y: 1010 × 1101

Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	**0**	**0**	**0**	**0**
1	0	0	0	**1**	**0**	**1**	**0**	0	0	**1**	**1**	**0**	0	0	0	**0**	**1**	**0**	**1**	**0**

Iter. 1: Com que **MR_0=1**, sumem P = P + MD
Desplacem MD a esquerra i MR a dreta

Multiplicació de naturals

- Exemple. Multipliquem x × y: 1010 × 1101

Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	**0**	**0**	**0**	**0**
1	0	0	0	**1**	**0**	**1**	**0**	0	0	**1**	**1**	**0**	0	0	0	**0**	**1**	**0**	**1**	**0**
2													0	0	**0**	**0**	**1**	**0**	**1**	**0**

Iter. 2: Com que $\mathbf{MR_0=0}$, P no es modifica

Multiplicació de naturals

- Exemple. Multipliquem x × y: 1010 × 1101

Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	**0**	**0**	**0**	**0**
1	0	0	0	**1**	**0**	**1**	**0**	0	0	**1**	**1**	**0**	0	0	0	**0**	**1**	**0**	**1**	**0**
2	0	0	**1**	**0**	**1**	**0**	0	0	0	0	**1**	**1**	0	0	**0**	**0**	**1**	**0**	**1**	**0**

Iter. 2: Com que **MR_0=0**, P no es modifica
Desplacem MD a esquerra i MR a dreta

Multiplicació de naturals

- Exemple. Multipliquem x × y: 1010 × 1101

Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	**0**	**0**	**0**	**0**
1	0	0	0	**1**	**0**	**1**	**0**	0	0	**1**	**1**	**0**	0	0	0	**0**	**1**	**0**	**1**	**0**
2	0	0	**1**	**0**	**1**	**0**	0	0	0	0	**1**	**1**	0	0	**0**	**0**	**1**	**0**	**1**	**0**
3												**+**	0	0	**1**	**0**	**1**	**0**	0	0

Iter. 3: Com que **MR_0=1**, sumem P = P + MD

Multiplicació de naturals

- Exemple. Multipliquem x × y: 1010 × 1101

Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	**0**	**0**	**0**	**0**
1	0	0	0	**1**	**0**	**1**	**0**	0	0	**1**	**1**	**0**	0	0	0	**0**	**1**	**0**	**1**	**0**
2	0	0	**1**	**0**	**1**	**0**	0	0	0	0	**1**	**1**	0	0	**0**	**0**	**1**	**0**	**1**	**0**
3	0	**1**	**0**	**1**	**0**	0	0	0	0	0	0	**1**	0	**0**	**1**	**1**	**0**	**0**	**1**	**0**

Iter. 3: Com que **MR_0=1**, sumem P = P + MD
Desplacem MD a esquerra i MR a dreta

Multiplicació de naturals

- Exemple. Multipliquem x × y: 1010 × 1101

Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0
1	0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	0	1	0	1	0
2	0	0	1	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0
3	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0
4												+	0	1	0	1	0	0	0	0

Iter. 4: Com que **MR_0=1**, sumem P = P + MD

Multiplicació de naturals

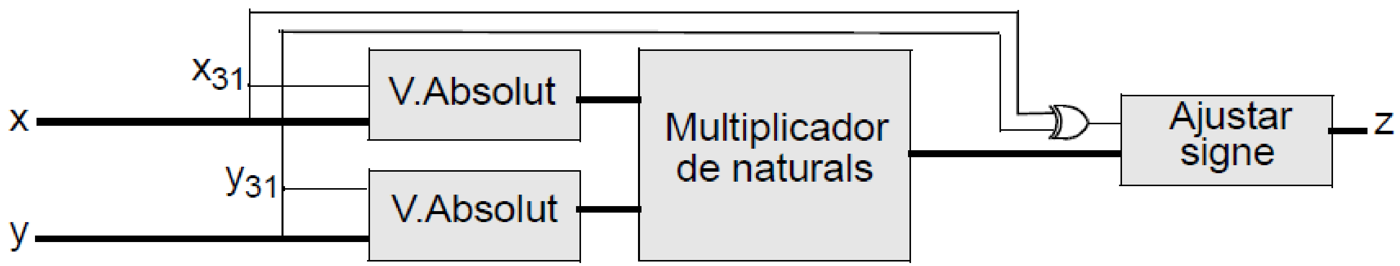
- Exemple. Multipliquem x × y: 1010 × 1101

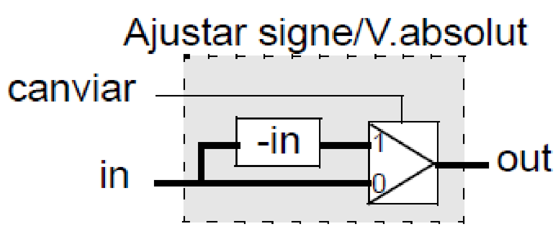
Iter.	MD (Multiplicand)								MR (Multiplicador)				P (Producte)							
Init	0	0	0	0	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	0	0	0	0	**0**	**0**	**0**	**0**
1	0	0	0	**1**	**0**	**1**	**0**	0	0	**1**	**1**	**0**	0	0	0	**0**	**1**	**0**	**1**	**0**
2	0	0	**1**	**0**	**1**	**0**	0	0	0	0	**1**	**1**	0	0	**0**	**0**	**1**	**0**	**1**	**0**
3	0	**1**	**0**	**1**	**0**	0	0	0	0	0	0	**1**	0	**0**	**1**	**1**	**0**	**0**	**1**	**0**
4	**1**	**0**	**1**	**0**	0	0	0	0	0	0	0	0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

Iter. 4: Com que **MR_0=1**, sumem P = P + MD
Desplacem MD a esquerra i MR a dreta

Multiplicació d'enters

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





Multiplicació d'enters

- Instruccions MIPS

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

Multiplicació d'enters

- Instruccions MIPS

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

- `$hi` i `$lo` són registres especials
  - No es poden usar a la resta d'instruccions estudiades fins ara

Multiplicació d'enters

- Instruccions MIPS

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

- `$hi` i `$lo` són registres especials
  - No es poden usar a la resta d'instruccions estudiades fins ara
- Per moure el resultat a registres de propòsit general:

```
mflo     rd              # rd ← $lo
mfhi     rd              # rd ← $hi
```

Multiplicació d'enters

- Instruccions MIPS

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

- `$hi` i `$lo` són registres especials
  - No es poden usar a la resta d'instruccions estudiades fins ara
- Per moure el resultat a registres de propòsit general:

```
mflo     rd          # rd ← $lo
mfhi     rd          # rd ← $hi
```

- Overflow
  - Naturals: si `$hi ≠ 0`
  - Enters: si `$hi:$lo` no és l'extensió de signe de `$lo`

Divisió de naturals

- En base 10: 421 / 13

```
                       421   |013
Prova 1:          ≱    013   ¯¯¯¯¯¯   --> Hi cap a 0, multiplicar: 0×013=000
  (restar 0)      -    000
                      ----
                       421
```

Divisió de naturals

- En base 10

```
                               421   |013
                                     -------
Prova 1:          ≱         013           --> Hi cap a 0, multiplicar: 0×013=000
  (restar 0)      –         000
                           -----
                               421
Prova 2:          ≥          013          --> Hi cap a 3, multiplicar: 3×013=039
  (restar)        –          039
                            -----
                               031
```

Divisió de naturals

- En base 10

```
                               421   |013
Prova 1:          ≱          013    ---------   --> Hi cap a 0, multiplicar: 0×013=000
  (restar 0)       –          000
                             -----
                               421
Prova 2:          ≥           013               --> Hi cap a 3, multiplicar: 3×013=039
  (restar)         –           039
                             -----
                               031
Prova 3:          ≥            013              --> Hi cap a 2, multiplicar: 2×013=026
  (restar)         –            026
                             -----
                               005
```

Divisió de naturals

- En base 10

```
                           421   |013
                                 ------
Prova 1:           ≱      013            --> Hi cap a 0, multiplicar: 0×013=000
  (restar 0)       -      000
                          ----
                           421
Prova 2:           ≥       013           --> Hi cap a 3, multiplicar: 3×013=039
  (restar)         -       039
                          ----
                           031
Prova 3:           ≥       013           --> Hi cap a 2, multiplicar: 2×013=026
  (restar)         -       026
                          ----
                           005           --> Residu = 005, Quocient = 032
```

Divisió de naturals

- En base 2: 1011 / 0010

```
            Dividend =    1 0 1 1 | 0 0 1 0        = Divisor
prova 1:           ≱  0 0 1 0    --> 0             = Quocient
```

Divisió de naturals

- En base 2: 1011 / 0010

```
               Dividend =    1 0 1 1  | 0 0 1 0         = Divisor
prova 1:                 ≱  0 0 1 0   -> 0              = Quocient
     (no canvia)             1 0 1 1
```

Divisió de naturals

- En base 2: 1011 / 0010

```
            Dividend =    1 0 1 1  | 0 0 1 0       = Divisor
prova 1:         ≱  0 0 1 0          0  1          = Quocient
     (no canvia)          1 0 1 1
prova 2:         ≥    0 0 1 0
```

Divisió de naturals

- En base 2:   1011 / 0010

```
                 Dividend =     1 0 1 1 | 0 0 1 0        = Divisor
prova 1:                  ≱  0 0 1 0       0   1         = Quocient
      (no canvia)               1 0 1 1
prova 2:                  ≥    0 0 1 0
      (restar)               - 0 0 1 0
                               -------
                                0 0 1 1
```

Divisió de naturals

- En base 2: 1011 / 0010

```
                  Dividend =     1 0 1 1  | 0 0 1 0        = Divisor
prova 1:                 ≱  0 0 1 0         0   1   0      = Quocient
        (no canvia)              1 0 1 1
prova 2:                 ≥    0 0 1 0
        (restar)            - 0 0 1 0
                                 0 0 1 1
prova 3:                 ≱       0 0 1 0
```

Divisió de naturals

- En base 2: 1011 / 0010

```
            Dividend =     1 0 1 1  | 0 0 1 0         = Divisor
prova 1:              ≱  0 0 1 0      0  1  0         = Quocient
     (no canvia)                1 0 1 1
prova 2:              ≥    0 0 1 0
     (restar)            - 0 0 1 0
                                0 0 1 1
prova 3:              ≱      0 0 1 0
     (no canvia)                0 0 1 1
```

Divisió de naturals

- En base 2: 1011 / 0010

```
             Dividend =    1 0 1 1  | 0 0 1 0        = Divisor
prova 1:           ≱   0 0 1 0        0  1  0  1     = Quocient
      (no canvia)           1 0 1 1
prova 2:           ≥     0 0 1 0
      (restar)          - 0 0 1 0
                            0 0 1 1
prova 3:           ≱       0 0 1 0
      (no canvia)           0 0 1 1
prova 4:           ≥        0 0 1 0
```

Divisió de naturals

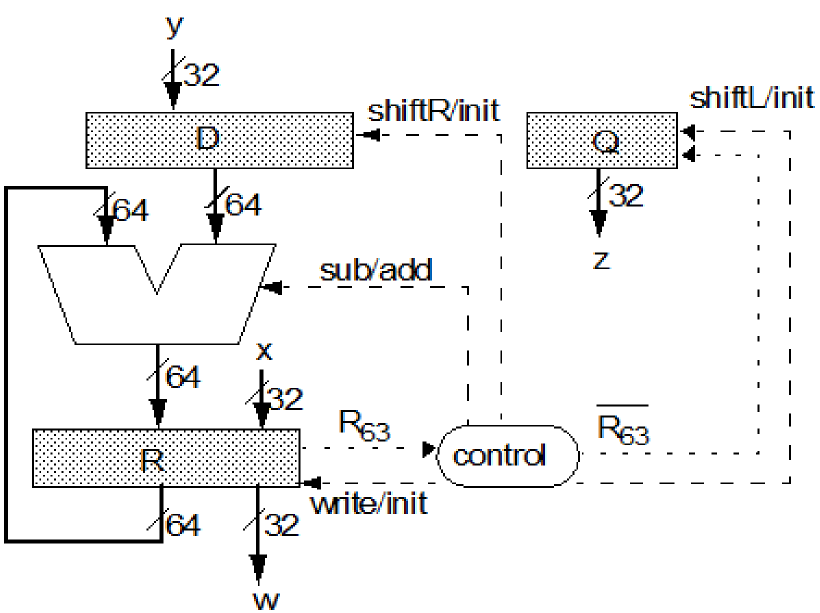
- En base 2: 1011 / 0010

```
                 Dividend =     1 0 1 1  | 0 0 1 0        = Divisor
prova 1:              ≱   0 0 1 0        +-----------
                                            0  1  0  1    = Quocient
      (no canvia)               1 0 1 1
prova 2:              ≥     0 0 1 0
      (restar)            - 0 0 1 0
                          ---------
                                0 0 1 1
prova 3:              ≱       0 0 1 0
                                -------
      (no canvia)               0 0 1 1
prova 4:              ≥         0 0 1 0
      (restar)                - 0 0 1 0
                                -------
                                0 0 0 1       = Residu
```

Divisió de naturals: circuit

- Divisió de naturals de 32 bits "amb restauració"
  - Quocient: $z = x / y$
  - Residu: $w = x \% y$

Divisor seqüencial de naturals

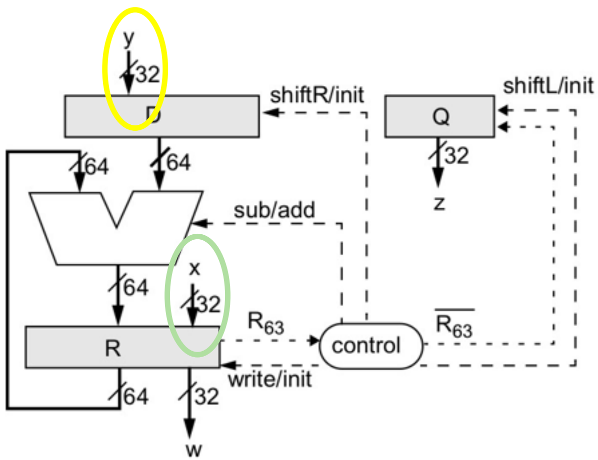


Pseudocodi

```
// Inicialització
R0:31 = x; R32:63 = 0;
D0:31 = 0; D32:63 = y;
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;
      }
}
z = Q;
w = R0:31
```

Exemple divisió x/y: 1011 / 0010

Init: ***x*** a la part baixa del Dividend (**R**), ***y*** a la part alta del Divisor (**D**), i zeros a la resta de bits

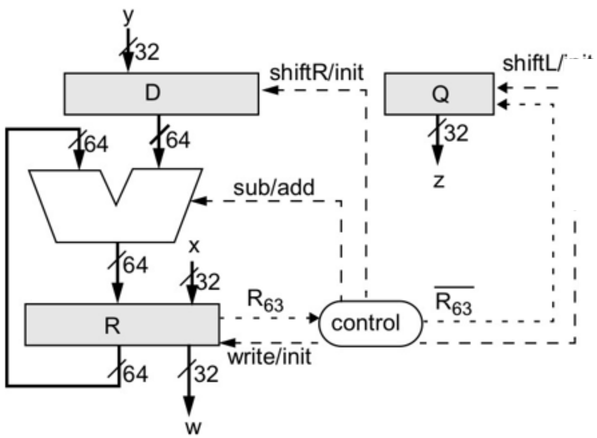



```
// Inicialització
R0:31 = x; R32:63 = 0;
D0:31 = 0; D32:63 = y;
Q  = 0;
```

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

Exemple divisió x/y: 1011 / 0010

Iter 1: Desplacem D a la dreta. Restem **R=R-D** per comparar-los. Comprovem que **R < D**!



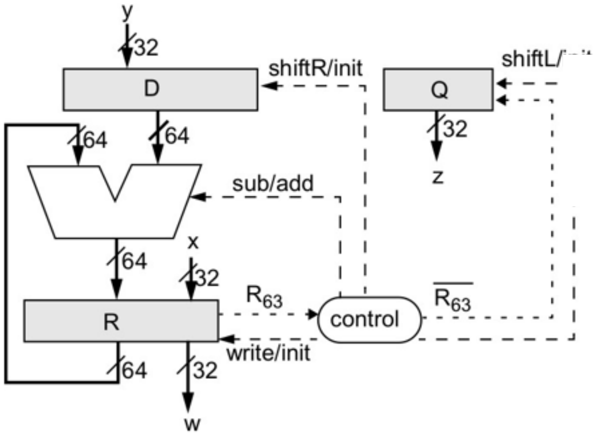

```
D = D >> 1;
R = R - D;
if (R63 == 0)
```

R	0	0	0	0	**1**	**0**	**1**	**1**
– D	0	**0**	**0**	**1**	**0**	0	0	0
	1	1	1	1	1	0	1	1

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	1	1	1	1	1	0	1	1	0	**0**	**0**	**1**	**0**	0	0	0	0	0	0	0

Exemple divisió x/y: 1011 / 0010

Iter 1: Restaurem R sumant **R=R+D**. Insertem **Q_0 = 0**



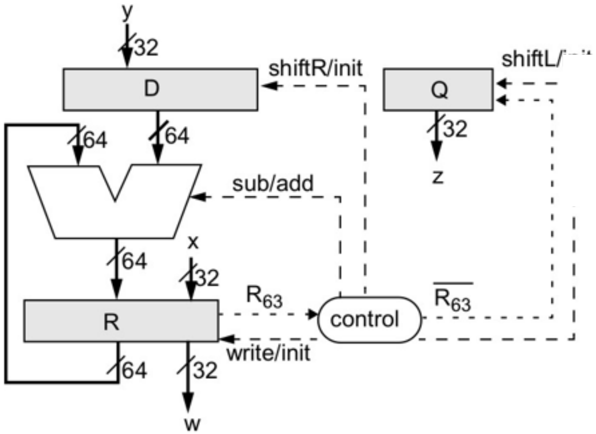

```
D = D >> 1;
R = R - D;
if (R63 == 0)

else {
        R = R + D;
        Q = Q << 1;
}
```

	R	1	1	1	1	1	0	1	1
+	D	0	**0**	**0**	**1**	**0**	0	0	0
		0	0	0	0	**1**	**0**	**1**	**1**

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**

Exemple divisió x/y: 1011 / 0010

Iter 2: Desplacem D a la dreta. Restem **R=R-D** i comprovem que **R ≥ D**. Insertem **Q_0 = 1**



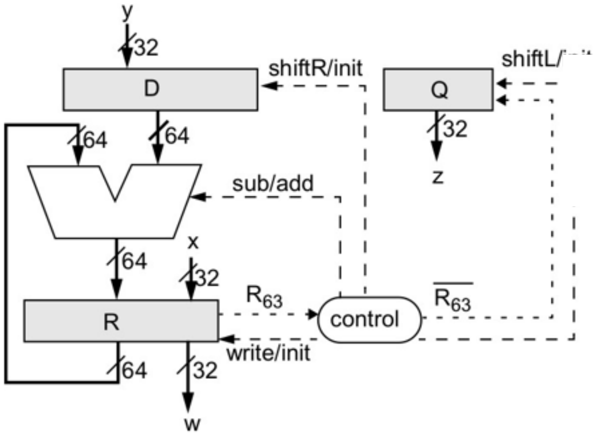

```
D = D >> 1;
R = R - D;
if (R63 == 0)
   Q = (Q << 1) | 1;
```

R	0	0	0	0	**1**	**0**	**1**	**1**
– D	0	0	**0**	**0**	**1**	**0**	0	0
	0	0	0	0	**0**	**0**	**1**	**1**

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**

Exemple divisió x/y: 1011 / 0010

Iter 3: Desplacem D a la dreta. Restem **R=R-D** per comparar-los. Comprovem que **R** < **D**!



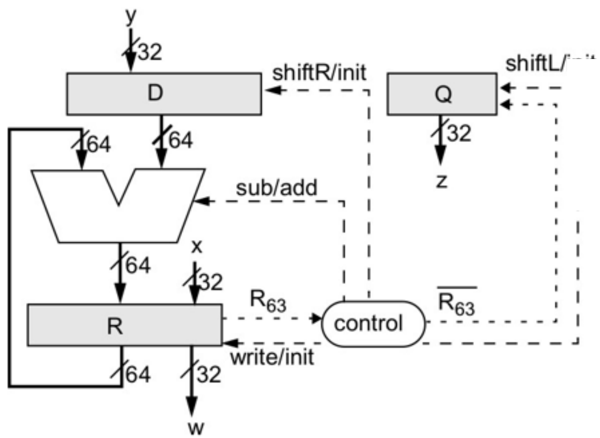

```
D = D >> 1;
R = R - D;
if (R63 == 0)
```

	R	0	0	0	0	**0**	**0**	**1**	**1**
–	D	0	0	0	**0**	**0**	**1**	**0**	0
		1	1	1	1	1	1	1	1

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	1	1	1	1	1	1	1	1	0	0	0	**0**	**0**	**1**	**0**	0	0	0	**0**	**1**

Exemple divisió x/y: 1011 / 0010

Iter 3: Restaurem R sumant **R=R+D**. Insertem $\mathbf{Q_0 = 0}$



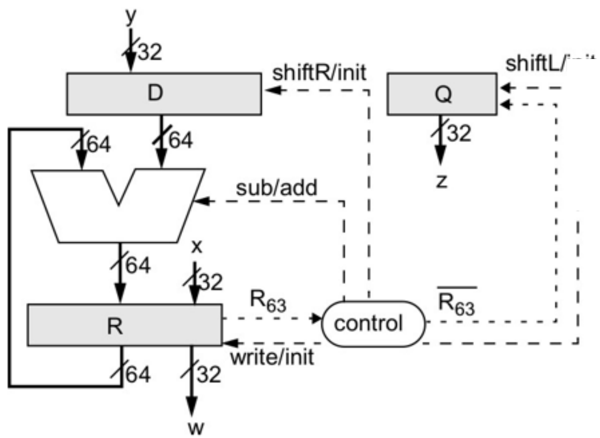

```
D = D >> 1;
R = R - D;
if (R63 == 0)

else {
        R = R + D;
        Q = Q << 1;
}
```

R	1	1	1	1	1	1	1	1
+ D	0	0	0	**0**	**0**	**1**	**0**	0
	0	0	0	0	**0**	**0**	**1**	**1**

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**

Exemple divisió x/y: 1011 / 0010

Iter 4: Desplacem D a la dreta. Restem **R=R-D** i comprovem que **R ≥ D**. Insertem **Q_0 = 1**




```
D = D >> 1;
R = R - D;
if (R63 == 0)
   Q = (Q << 1) | 1;
```

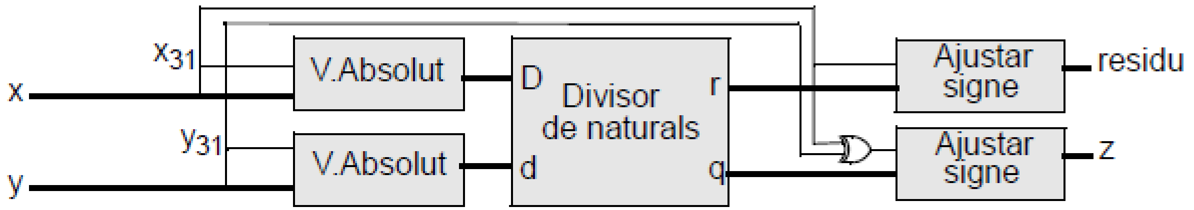
R	0	0	0	0	**0**	**0**	**1**	**1**
– D	0	0	0	0	**0**	**0**	**1**	**0**
	0	0	0	0	**0**	**0**	**0**	**1**

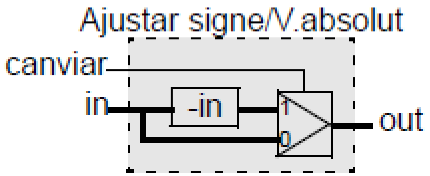
Iter.	R (Dividend/Residu)								D (Divisor)								Q (Quocient)			
Init	0	0	0	0	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	0	0	0	0	0	0	0	0
1	0	0	0	0	**1**	**0**	**1**	**1**	0	**0**	**0**	**1**	**0**	0	0	0	0	0	0	**0**
2	0	0	0	0	**0**	**0**	**1**	**1**	0	0	**0**	**0**	**1**	**0**	0	0	0	0	**0**	**1**
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**

Divisió d'enters: *x*/*y*

Si y ≠ 0:

1. Calcular valors absoluts del operands: $|x|$, $|y|$
2. Dividir els valors absoluts (naturals)
   - Obtenim quocient i residu
3. Ajustar signes
   - Canviar signe del quocient si x, y de diferent signe
   - Canviar signe del residu si el dividend és negatiu





Divisió d'enters i naturals

- Instruccions MIPS

```
divu    rs, rt          # Naturals
div     rs, rt          # Enters
```

- Operació

  $\$lo \leftarrow rs / rt$

  $\$hi \leftarrow rs \% rt$

- Si el divisor és 0, el resultat és *indefinit*
- Overflow
  - Naturals: no n'hi ha
  - Enters: només hi ha un cas. Quin és?

Divisió d'enters i naturals

- Instruccions MIPS

```
divu    rs, rt          # Naturals
div     rs, rt          # Enters
```

- Operació

  $\$lo \leftarrow rs\ /\ rt$

  $\$hi \leftarrow rs\ \%\ rt$

- Si el divisor és 0, el resultat és *indefinit*
- Overflow
  - Naturals: no n'hi ha
  - Enters: només hi ha un cas. Quin és?

$(-\,2^{31})\ /\ (-1) = 2^{31}$ → No representable!

Divisió per potències de 2

- **Algunes** divisions per potències de 2 es poden traduir per desplaçaments de bits
  - Molt més ràpids d'executar
- Per a naturals
  - Les instruccions `srl` i `divu` calculen el mateix quocient
- Però atenció! per a enters
  - Si el dividend és positiu, `sra` i `div` calculen el mateix quocient
  - Si el dividend és negatiu i la divisió no és exacta, `sra` i `div` donen resultats diferents
- En general
  - Traduïm els operadors de divisió (/) i mòdul (%) amb `div` i `divu`
  - Només optimitzarem amb `sra` si el resultat és equivalent a `div`

Aritmètica de coma flotant (primera part)

- Estàndard IEEE-754
  - Rang, precisió, arrodoniment
  - Codificacions especials, underflow, denormals
- Conversió de base 10 a base 2 i viceversa

Coma fixa

- **Coma fixa**
  - Alguns bits per a la part entera, i alguns per a la part fraccionària
  - Exemple amb 8 bits:

    `eeeeefff` → part entera i part fraccionària

    `10101,110`

$= 1\times2^{4} + 0\times2^{3} + 1\times2^{2} + 0\times2^{1} + 1\times2^{0} + 1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3}$

$= 21{,}75_{10}$

  - El rang és bastant limitat
  - Els números representables són equidistants

Coma flotant

- Rang molt més gran
  - però números representables no equidistants
- Notació científica normalitzada (base 10):

$v = \pm\ m \times 10^{e}$ tal que $1 \leq m < 10$

2.34×10^{6} → Normalitzt ($1 \leq m < 10$)

0.0234×10^{8} → No normalitzat!

(la part entera de m ha de tenir 1 sol dígit, i ha de ser no-nul)

Coma flotant en base 2

$$v = \underbrace{\pm}_{\text{signe (S)}} \underbrace{1,\overbrace{fff\ldots f}^{\text{fracció (F)}}}_{\text{mantissa}} \times \underbrace{2}_{\text{base}}{}^{\underbrace{eee\ldots e}_{\text{exponent (E)}}}$$

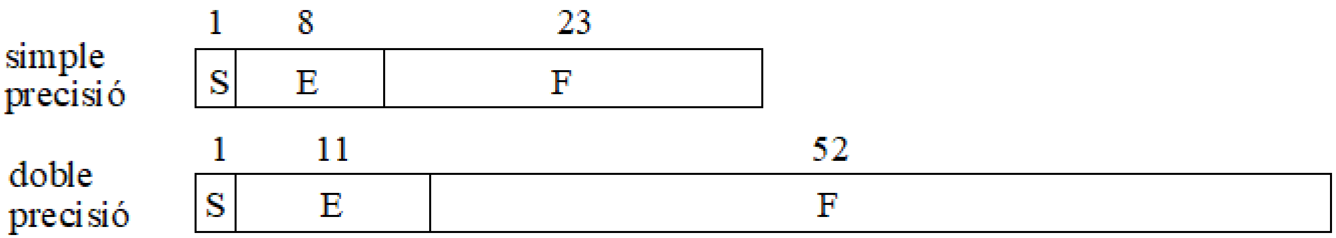
- Components:
  - **Signe**: 0=positiu, 1=negatiu
  - **Mantissa**: conté la magnitud del número
    - Normalitzada: part entera = 1 !
    - Part entera implícita ("bit ocult") → estalviem 1 bit
  - **Exponent**: enter representat "en excés"
  - **Base**: 2, implícita
- Codificació (32 bits): signe, exponent, fracció

```
v = seeeeeeeefffffffffffffffffffffff
```

- Interpretació (`S=s, E=eee...e - excés, F=0,fff...f`):

$$v = (-1)^{S} \times (1 + F) \times 2^{E}$$

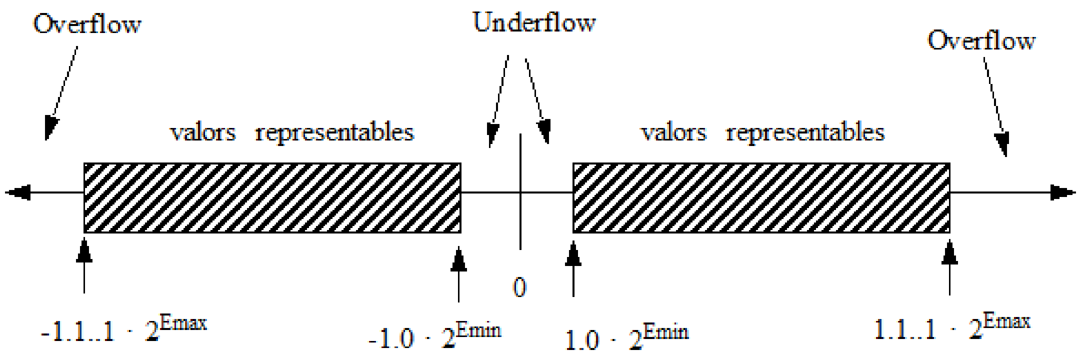
Estàndard IEEE-754 (1985-2019)



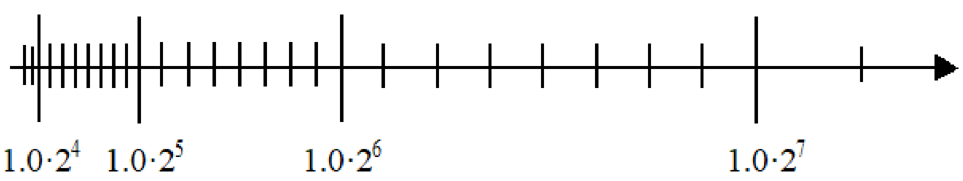
- Compromís rang vs. precisió
  - A més bits d'exponent: +rang, -precisió (i viceversa)
- Part entera = 1 (implícita, no es representa)
- Exponent
  - situat entre signe i fracció
  - en excés a 127 (simple precisió) o a 1023 (doble precisió)
  - Permet comparar magnituds amb un comparador de naturals
- En C, es representen amb els tipus `float` i `double`

IEEE-754: Valors representables

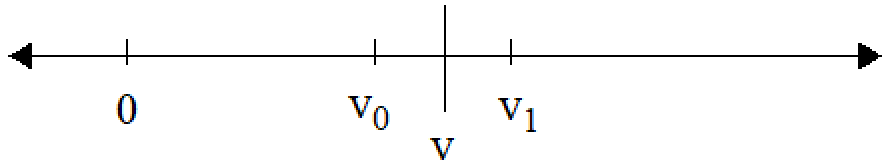
- Mantissa: `1,000...0` $\leq$ mantissa $\leq$ `1,111...1`
- Exponent: `Emin` $\leq$ E $\leq$ `Emax`
- Valors fora de rang: Overflow
- Valors amb magnitud < $1.0 \cdot 2^{Emin}$: Underflow (resultats "no fiables")



- Amb 32 bits es poden representar 2^{32} números, igual que amb coma fixa, però no són equidistants:



Error de precisió per arrodoniment



- Resultat exacte v està entre 2 valors representables: v_0 i v_1
- Si l'arrodonim a v_0 l'error de precisió és $\varepsilon = |v - v_0|$
- Exemple: representar el número racional $\frac{1}{10}$ en simple precisió
  - És el número amb fracció periòdica:

    $$v = 1{,}\underbrace{10011001100110011001100}_{\text{23 bits de fracció}}1100110011001100110011\ldots \times 2^{-4}$$

  - Si l'arrodonim a v_0 (eliminant bits)

    $$v_0 = 1{,}10011001100110011001100\cancel{1100110011001100110011}\ldots \times 2^{-4}$$

  - Error de precisió ($\varepsilon = |v - v_0|$)

    $$\varepsilon = 0{,}00000000000000000000000 1100110011001100110011\ldots \times 2^{-4}$$

IEE-754: 4 modes d’arrodoniment

En l’exemple anterior…

$$v_0 = 1{,}1001100110011001100110\mathbf{0} \times 2^{-4}$$

$$v = 1{,}10011001100110011001100\color{red}{11001100110011001100110011\ldots} \times 2^{-4}$$

$$v_1 = 1{,}1001100110011001100110\mathbf{1} \times 2^{-4}$$

1.- **Truncament** (“cap al zero”): $v=v_0$

- El més simple d’implementar: eliminant els bits extra
- Cota superior d’error (quan v està pròxim a v_1):

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

$$= 0{,}0000000000000000000000\mathbf{1} \times 2^{-4} = 2^{-4-23}$$

= 1 ULP (*Unit in the Last Place*)

2.- **Cap al +∞**: $v=\max(v_0,v_1)$

3.- **Cap al - ∞** : $v=\min(v_0,v_1)$

- Usats en “aritmètica d’intervals”

IEE-754: 4 modes d'arrodoniment

En l'exemple anterior...

v_0 = 1,10011001100110011001100 × 2^{-4}

v = 1,1001100110011001100110011001100110011001100110011… × 2^{-4}

v_1 = 1,10011001100110011001101 × 2^{-4}

4.- **Cap al més pròxim** (o al valor "parell" en el cas equidistant)

- Mètode **usat per defecte**
  - → En l'exemple, arrodonim a v_1, que és més pròxim que v_0
- Màxim error: quan v és equidistant de v_0 i v_1

  $\varepsilon_{max} = |v_1 - v_0|/2 = 0{,}5$ ULP
- **Regla pràctica**: examinem el primer bit extra en 3 exemples

v = 1,xxxxxx...0011 $\boxed{0}$000101 → v_0 = 1,xxxxxx...0011

v = 1,xxxxxx...0011 $\boxed{1}$000101 → 1,xxxxxx...0011

+ 0,000000...0001

v_1 = 1,xxxxxx...0100

v = 1,xxxxxx...0011 $\boxed{1000000..}$ → al parell = v_0: 1,xxxxxx...010$\boxed{0}$

Equidistant!

IEE-754: Codificacions especials

- Exponents en valors normalitzats: `Emin= 0..001`, `Emax = 1..110`
- Exponents reservats: `000..0` i `111..1`
- **±Zero** (`E=000..0, F=000..0`)

  No es pot representar en format normalitzat...

  Pega: tenim un zero positiu i un negatiu!
- **Denormals** (`E=000..0, F≠000..0`)

  Bit ocult implícit = `0` (en lloc de `1`). Exponent implícit = `Emin`

  Representen números en la "regió d'*underflow*": $0,xxx..x \times 2^{Emin}$
- ± **Inf** (`E=111..1, F=000..0`)

  Evitar *overflows* en càlculs intermedis : `1/(1+100/x)=0` per a $x \to 0$

  Definim algunes regles: `1÷0=Inf` , `1/Inf=0`, `x±Inf=Inf`, etc.
- **NaN** (*Not-a-Number*) (`E=111..1, F≠000..0`)

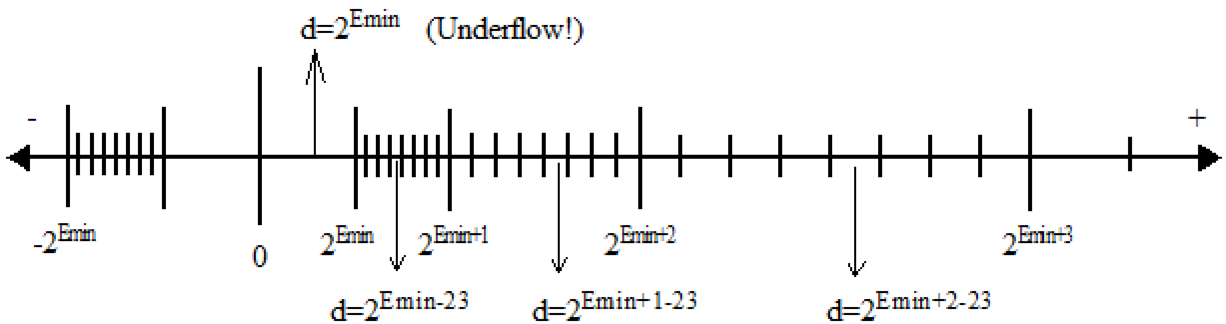
  Resultat d'operacions *invàlides*: $\infty \pm \infty$, $0 \times \infty$, $\infty \div \infty$, $\sqrt{x}\ (x<0)$, etc.

  Definim algunes regles: `x operat NaN = NaN`

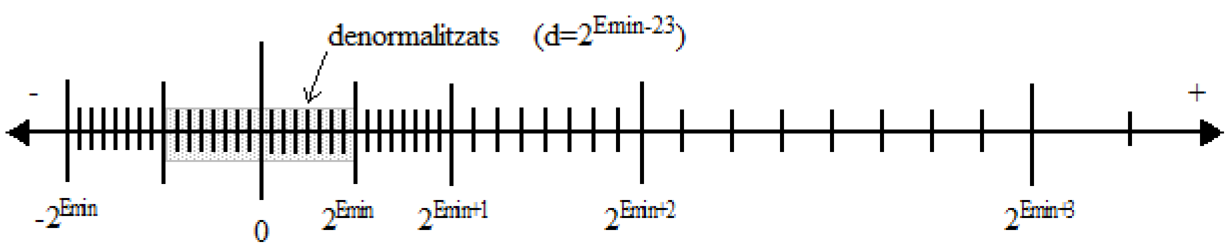
IEE-754: resum dels formats

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

Underflow i nombres denormals



- L'error de precisió depèn de la distància (d) entre números consecutius
- No és uniforme: decreix geomètricament amb l'exponent…
- Excepte prop del zero!
- Prop del zero, l'error és del mateix ordre que la magnitud (l'error relatiu pot arribar a ser = 1). Els resultats no són fiables: Excepció d'Underflow
- Alternativa: números denormals: `0,xxxx...xx × 2^Emin`



Exemple. Representar `v=-1029,68`

1. Convertir la part entera, per divisions successives

   `1029 = 10000000101`   (té `11` bits)

2. Convertir la fracció, per multiplicacions successives

   `0,68 × 2 = `**`1`**`,36      → 1   (primer bit de fracció)`

   `0,36 × 2 = `**`0`**`,72      → 0   (segon bit de fracció)`

   `...` etc.

   - Seguim fins tenir 24 bits entre part entera i fracció: `24-11=13` bits

   - Però calen alguns bits extra, suficients per discernir si arrodonir amunt o al cas equidistant (al parell). En aquest cas ens calen 5 bits extra: **`10001`**...

   `0,68 = 0,1010111000010`**`10001`**`...`   (`18` bits)

3. Ajuntar part entera i fracció (`11+18=29` bits)

   `1029,68 = 10000000101,1010111000010`**`10001`**`...`

Exemple (cont.)

4. Normalitzar (moure la coma `10` llocs i ajustar l'exponent)
   `1029,68 = 1,000000010110101110000101`**`10001`**`... ×` 2^{10}
5. Arrodonir (al més pròxim), examinant els "bits extra" → amunt
   `1029,68 = 1,000000010110101110000101`**`10001`**`... ×` 2^{10}
   `1029,68 = 1,00000001011010111000001`**`1`**` ×` 2^{10}
6. Codificar l'exponent en excés a `127`
   `10 + 127 = 137 = 10001001`
7. Ajuntar Signe, Exponent i Fracció (el bit implícit `1` no s'escriu!)
   `-1029,68 = 1 10001001 00000001011010111000011`
8. Expressar en hexadecimal
   `-1029,68 =` **`0xC480B5C3`**

Exemple (cont.)

- Calcular l'error de precisió ($\varepsilon = |\mathtt{v}-\mathtt{v}_0|$)
  - Restant el valor arrodonit i el valor "exacte"

$$
\begin{array}{rlr}
\varepsilon = & (\ \ \mathtt{1,00000001011010111000011} & \times\ 2^{10} \\
 & -\ \underline{\mathtt{1,00000001011010111000010}\textcolor{red}{\mathbf{10001}}\mathtt{...}} & \times\ 2^{10}) \\
= & \mathtt{0,0000000000000000000000001111} & \times\ 2^{10}
\end{array}
$$

  - Normalitzem: movem la coma 25 posicions a la dreta …

$$\varepsilon = 1,111 \times 2^{10-25} = \mathbf{1,111 \times 2^{-15}}$$

  - Convertim a decimal (no es demanarà sense calculadora) …

$$
\begin{aligned}
\varepsilon &= 1,875 \times 2^{-15} \\
&= 1,875 / 2^{15} \\
&= 1,875/32768 \\
&= \mathbf{5,722 \times 10^{-5}}
\end{aligned}
$$

Exemple: convertir `v=0x45814140` a base `10`

1. L'escrivim en binari
   `v = 0100 0101 1000 0001 0100 0001 0100 0000`
2. Agrupem els bits en camps (Signe, Exponent, Fracció)
   `v = 0 10001011 00000010100000101000000`
3. Convertim l'exponent a decimal i li restem l'excés `127`
   `10001011 = 139`
   `E = 139 - 127 = 12`
4. Ajuntar signe, part entera (bit ocult), fracció i potència
   $v = +\mathbf{1},00000010100000101000000 \times 2^{12}$
5. Moure la coma `12` posicions a la dreta i eliminar zeros finals
   $v = +1000000101000,00101\cancel{000000}$
6. Convertir la part entera a decimal (suma ponderada)
   $1000000101000 = 1 \cdot 2^{12} + 1 \cdot 2^{5} + 1 \cdot 2^{3} = 4136$
7. Convertir la fracció a decimal (movent la coma a la dreta)
   $0,00101 = 101 \times 2^{-5} = 5/32 = 0,15625$
8. Ajuntar part entera i fracció: **`v = 4136,15625`**

Aritmètica de coma flotant (segona part)

- Operacions
  - Bits de guarda
  - Suma (resta)
  - Multiplicació i divisió
- Coma flotant en MIPS
- Associativitat de la suma

Bits de guarda

- L'estàndard IEEE-754 especifica que *el resultat de una operació aritmètica ha de ser el mateix que el que s'obtindria si es realitzés amb una precisió absoluta i al final s'arrodonís el resultat* (al més pròxim → ε_{max} = 0,5 ULP)
- El resultat intermedi d'una operació pot tenir molts (més de 250!) bits "extra"

```
y = X,XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX...
```

  – Operar amb tots ells per no perdre precisió requeriria un hardware monstruós!
  – Però es demostra que podem obtener el mateix resultat convertint el conjunt de bits extra en tan sols tres (anomenats *bits de guarda*): **G**uard, **R**ound i **S**ticky
- Per convertir el conjunt de bits extra a 3 bits de guarda:
  – **G** i **R** són iguals als 2 primers bits extra
  – **S** (tercer bit) és la OR lógica de la resta de bits a la dreta
- Suposem el resultat intermedi

```
y = X,XXXXXXXXXXXXXXXXXXXXXXXabcdefgh...
```

- Fem: **G**=**a**, **R**=**b**, **S**=(**c** OR **d** OR **e** OR **f** OR **g** OR **h** ...)

```
y = X,XXXXXXXXXXXXXXXXXXXXXXXGRS
```

Suma (resta) en coma flotant

- Suposem un cas senzill que coneixem bé (base `10`):
  - Format: mantissa normalitzada amb 4 dígits decimals: $x,xxx \times 10^{xx}$
  - Sumar: $9,999 \times 10^{1} + 1,680 \times 10^{-1}$
  - Així no:
    $$\cancel{\begin{array}{rrl} & 9,999 & \times 10^{1} \\ + & 1,680 & \times 10^{\mathbf{-1}} \\ \hline = & 11,679 & \times 10^{?} \end{array}}$$
  - Alinear mantisses:
    $$\begin{array}{rrl} & 9,999 & \times 10^{1} \\ + & \mathbf{0,0}1680 & \times 10^{\mathbf{1}} \\ \hline = & 10,01580 & \times 10^{1} \end{array}$$
  - Normalitzar:
    $$1,001580 \times 10^{\mathbf{2}}$$
  - Arrodonir a 4 dígits:
    $$1,001\mathbf{580} \times 10^{2}$$
    $$\boxed{1,00\mathbf{2} \times 10^{2}}$$

Suma (resta) en coma flotant

1. **Igualar els exponents** al major dels dos, desplaçant la coma de la mantissa del de menor exponent
2. **Sumar les magnituds** tenint en compte els signes: si són diferents cal restar-les (en aquest cas, posar al minuend la de major valor absolut!)
3. **Normalitzar el resultat** movent la coma per obtenir un `1` a la part entera
4. **Arrodonir** la mantissa
5. **Codificar** signe, exponent (en excés) i mantissa (eliminant bit ocult)

Exemple: sumar z=x+y

Suposem `x=0x3F40000D`, `y=0xC0800004`

1. Els escrivim en binari

   `x = 0011 1111 0100 0000 0000 0000 0000 1101`

   `y = 1100 0000 1000 0000 0000 0000 0000 0100`

2. Separem els camps (S, E, F)

   `x = 0 01111110 10000000000000000001101`

   `y = 1 10000001 00000000000000000000100`

3. Convertim exponents a decimal, afegim bit ocult, i el signe

   x = **+1,**10000000000000000001101 × 2^{-1}

   y = **-1,**00000000000000000000100 × 2^{2}

4. Igualem exponents al major (=2): desplacem x tres bits a la dreta

   x = **+**0,00**1**10000000000000000001**101** × 2^{2}

   bits extra

Exemple (cont.)

5. Sumar magnituds: signes diferents → restar (minuend, la major = |y|)

```
                                          GRS
 |y|  = 1,00000000000000000000100              × 2^2
-|x|  = 0,00110000000000000000001 101          × 2^2
 |z|  = 0,11010000000000000000010 011          × 2^2
```

6. Normalitzar mantissa (desplaçant els bits 1 posició a l'esquerra)

```
 |z| = 1,10100000000000000000100 11  × 2^1
```

7. Arrodonir mantissa (amunt)

```
       1,10100000000000000000100  × 2^1
     +                         1
 |z| = 1,10100000000000000000101  × 2^1
```

8. Ajuntar signe, exponent (en excés) i mantissa (sense el bit ocult)

exponent = 1 + 127 = 128 = 10000000

signe = 1

```
z = 1 10000000 10100000000000000000101
  = 1100 0000 0101 0000 0000 0000 0000 0101
  = 0xC0500005
```

Multiplicació (divisió) en coma flotant

- Observem que

$$m_1 \times 2^{e1} \times m_2 \times 2^{e2} = (m_1 \times m_2) \times 2^{(e1+e2)}$$

$$m_1 \times 2^{e1} / m_2 \times 2^{e2} = (m_1 / m_2) \times 2^{(e1-e2)}$$

- Passos
  - Multiplicar (o dividir) les mantisses (igual que amb els naturals)
  - Sumar exponents (o restar-los)

Multiplicació en coma flotant

- Suposem un cas senzill que coneixem bé (base 10):
  - Format: mantissa normalitzada amb 4 dígits decimals: $x,xxx \times 10^{xx}$
  - Multiplicar: $(-1,110 \times 10^{10}) \times 9,200 \times 10^{-5}$
  - Suma dels exponents: $10 + (-5) = 5$
  - Producte de les mantisses:

$$
\begin{array}{r}
1,110 \\
\times \quad 9,200 \\
\hline
0000 \\
0000 \\
2220 \\
9990 \\
\hline
10,212000
\end{array}
$$

  - Ajuntar-ho tot: $-10,212000 \times 10^{5}$
  - Normalitzar: $-1,0212000 \times 10^{6}$
  - Arrodonir a 4 dígits (avall): $-1,021\mathbf{2000} \times 10^{6}$

$$\boxed{-1,02\mathbf{1} \times 10^{6}}$$

Exemple: multiplicar z=x×y

Suposem `x=0x3F600000`, `y=0xBED00002`

1. Els escrivim en binari
```
x = 0011 1111 0110 0000 0000 0000 0000 0000
y = 1011 1110 1101 0000 0000 0000 0000 0010
```
2. Separem els camps (S, E, F)
```
x = 0 01111110 11000000000000000000000
y = 1 01111101 10100000000000000000010
```
3. Convertim exponents a decimal, afegim bit ocult, i el signe
   - x = **+1**,11000000000000000000000 × 2^{-1}
   - y = **-1**,10100000000000000000010 × 2^{-2}
4. Multipliquem mantisses (sense zeros finals de `x`, parts enteres en negreta)

```
      1110...
 ×    110100000000000000000000010
      ---------------------------
                          1110...
      1110...
     1110...
    1110...
    ---------------------------
  101101100000000000000000011100...(+20 bits 0)
```

Exemple (cont.)

5. Sumem els exponents: `-1+(-2) = -3`
6. El producte queda **10,**11011000000000000000011**100** x 2^{-3}

   el Sticky és la or lògica

7. Normalitzem: desplacem tots els bits 1 posició a la dreta

   |z| = **1,**01101100000000000000001**110** × 2^{-2}

8. Arrodonim la mantissa (amunt)

```
         1,01101100000000000000001110  × 2^-2
      +                         1
|z| =    1,01101100000000000000010     × 2^-2
```

9. Codifiquem l'exponent: `-2+127 = 125 = 01111101`
10. Ajuntem signe (negatiu), exponent i mantissa (sense el bit ocult!)

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

Coma flotant en el MIPS

- Coprocesador de coma flotant (o FP) CP1
  - Procesador adjunt que extén el ISA
  - 32 registres de simple precisió: `$f0, $f1, ... $f31`
  - O bé 16 parells per a doble precisió: `$f0-$f1, $f2-$f3,...`
  - Les instruccions FP operen només sobre registres FP:
- Accés a memòria

```
lwc1  ft,offset(rs)      ldc1  ft,offset(rs)
swc1  ft,offset(rs)      sdc1  ft,offset(rs)
```

- Aritmètiques

```
add.s   fd,fs,ft         add.d   fd,fs,ft
sub.s   fd,fs,ft         sub.d   fd,fs,ft
mul.s   fd,fs,ft         mul.d   fd,fs,ft
div.s   fd,fs,ft         div.d   fd,fs,ft
```

Instruccions de coma flotant

- Còpia entre registres

`mfc1 rt,fs`	→	copia de `fs` a `rt`
`mtc1 rt,fs`	→	copia de `rt` a `fs`
`mov.s fd,fs`	→	copia de fs a fd

- Comparació

`c.xx.s fs,ft`	`c.xx.d fs,ft`

  - on `xx` és un de {`eq,lt,le`}
  - escriu el resultat al *bit de condició* (registre implícit)

- Salt

`bc1t etiqueta`	→	salta si el bit de condició = `TRUE`
`bc1f etiqueta`	→	salta si el bit de condició = `FALSE`

Subrutines i declaracions

- Pas de paràmetres i resultats a subrutines
  - Reglamentació molt complexa per al pas de `floats`, *`doubles`* i mescles de tipus enters amb flotants
  - Sols veurem el cas amb `1` o `2` paràmetres `float`: s'usen `$f12` i `$f14`
  - El resultat es retorna en `$f0`
- Registres "segurs": del `$f20` al `$f31`
- Declaració de variables de coma flotant en MIPS

```
       .data
var1:  .float 3.1416, 3.5E2,...
var2:  .double 3E350, 0.0038, ...
```

  - Alineeen a adreces múltiples de 4 i de 8, respectivament

Exemple: traduir la funció `func()`

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

```
        .data
const1: .float 1.0

        .text
        ...
func:
        la      $t0,const1
        lwc1    $f16,0($t0)     # $f16 = 1.0
        c.lt.s  $f12,$f16       # cond = (x<1.0)
        bc1f    else            # branch if cond=false
        mul.s   $f0,$f12,$f12   # x*x
        b fisi
else:
        add.s   $f16,$f16,$f16 # 1.0+1.0
        sub.s   $f0,$f16,$f12   # 2.0-x
fisi:
        jr $ra
```

Associativitat

- Verifica la propietat associativa la suma de números en coma flotant?

$$\boxed{x + (y + z) = (x + y) + z}$$

- Suposem $x = -1{,}5 \times 10^{38}$, $y = 1{,}5 \times 10^{38}$, $z = 1{,}0$

$x + (y + z) = -1{,}5 \times 10^{38} + (1{,}5 \times 10^{38} + 1{,}0)$

$= -1{,}5 \times 10^{38} + 1{,}5 \times 10^{38}$

$= 0{,}0$

$(x + y) + z = (-1{,}5 \times 10^{38} + 1{,}5 \times 10^{38}) + 1{,}0$

$= 0{,}0 + 1{,}0$

$= 1{,}0$

NO!

Exercici: sumar z=x+y

Suposem `x=3DC00046`, `y=0xC0800004`

Exercici: sumar z=x+y

Suposem `x=3DC00046`, `y=0xC0800004`

1. Els escrivim en binari

   ```
   x = 0011 1101 1100 0000 0000 0000 0100 0110
   y = 1100 0000 1000 0000 0000 0000 0000 0100
   ```

Exercici: sumar z=x+y

Suposem `x=3DC00046`, `y=0xC0800004`

1. Els escrivim en binari

```
x = 0011 1101 1100 0000 0000 0000 0100 0110
y = 1100 0000 1000 0000 0000 0000 0000 0100
```

2. Separem els camps (S, E, F)

```
x = 0 01111011 10000000000000001000110
y = 1 10000001 00000000000000000000100
```

Exercici: sumar z=x+y

Suposem `x=3DC00046`, `y=0xC0800004`

1. Els escrivim en binari

   `x = 0011 1101 1100 0000 0000 0000 0100 0110`

   `y = 1100 0000 1000 0000 0000 0000 0000 0100`

2. Separem els camps (S, E, F)

   `x = 0 01111011 10000000000000001000110`

   `y = 1 10000001 00000000000000000000100`

3. Convertim exponents a decimal, afegim bit ocult, i el signe

   x = **+1,**10000000000000001**000110** $\times 2^{-4}$

   y = **-1,**00000000000000000000100 $\times 2^{2}$

Exercici: sumar z=x+y

Suposem `x=3DC00046`, `y=0xC0800004`

1. Els escrivim en binari

   `x = 0011 1101 1100 0000 0000 0000 0100 0110`

   `y = 1100 0000 1000 0000 0000 0000 0000 0100`

2. Separem els camps (S, E, F)

   `x = 0 01111011 10000000000000001000110`

   `y = 1 10000001 00000000000000000000100`

3. Convertim exponents a decimal, afegim bit ocult, i el signe

   x = **+1,**10000000000000001**000110** × 2^{-4}

   y = **-1,**00000000000000000000100 × 2^{2}

4. Igualem exponents al major (=2): desplacem `x` sis bits a la dreta

   x = **+**0,00000**1**10000000000000001**000110** × $2^{\mathbf{2}}$

   bits extra

Exercici: sumar z=x+y

Suposem `x=3DC00046`, `y=0xC0800004`

1. Els escrivim en binari
   `x = 0011 1101 1100 0000 0000 0000 0100 0110`
   `y = 1100 0000 1000 0000 0000 0000 0000 0100`
2. Separem els camps (S, E, F)
   `x = 0 01111011 10000000000000001000110`
   `y = 1 10000001 00000000000000000000100`
3. Convertim exponents a decimal, afegim bit ocult, i el signe
   x = **+1**,10000000000000001**000110** × 2^{-4}
   y = **-1**,00000000000000000000100 × 2^{2}
4. Igualem exponents al major (=2): desplacem x sis bits a la dreta
   x = **+**0,00000**1**10000000000000001**000110** × 2^{2}
   bits extra
5. Determinem els 3 bits de guarda GRS
   x = **+**0,00000**1**10000000000000001**001** × 2^{2}

Exemple (cont.)

6. Sumar magnituds: signes diferents → restar (minuend, la major = |y|)

```
                                            GRS
  |y|   = 1,00000000000000000000100               × 2^2
- |x|   = 0,00000110000000000000001   001   × 2^2
  |z|   = 0,11111010000000000000010   111   × 2^2
```

Exemple (cont.)

6. Sumar magnituds: signes diferents → restar (minuend, la major = |y|)

GRS

|y| = 1,00000000000000000000100 $\times 2^2$

\- |x| = 0,00000110000000000000001**001** $\times 2^2$

|z| = 0,11111010000000000000010**111** $\times 2^2$

7. Normalitzar mantissa (desplaçant els bits 1 posició a l'esquerra)

|z| = **1**,11110100000000000000101**11** $\times 2^1$

Exemple (cont.)

6. Sumar magnituds: signes diferents → restar (minuend, la major = |y|)

```
                                               GRS
 |y|  = 1,00000000000000000000100                × 2^2
-|x|  = 0,00000110000000000000001 001            × 2^2
 |z|  = 0,11111010000000000000010 111            × 2^2
```

7. Normalitzar mantissa (desplaçant els bits 1 posició a l'esquerra)

```
 |z| = 1,11110100000000000000000101 11 × 2^1
```

8. Arrodonir mantissa (amunt)

```
       1,11110100000000000000000101 × 2^1
     +                            1
 |z| = 1,11110100000000000000000110 × 2^1
```

Exemple (cont.)

6. Sumar magnituds: signes diferents → restar (minuend, la major = |y|)

```
                                    GRS
  |y|  = 1,00000000000000000000100       × 2^2
- |x|  = 0,00000110000000000000001 001   × 2^2
  |z|  = 0,11111010000000000000010 111   × 2^2
```

7. Normalitzar mantissa (desplaçant els bits 1 posició a l'esquerra)

```
  |z| = 1,11110100000000000000101 11 × 2^1
```

8. Arrodonir mantissa (amunt)

```
        1,11110100000000000000101 × 2^1
      +                         1
  |z| = 1,11110100000000000000110 × 2^1
```

9. Ajuntar signe, exponent (en excés) i mantissa (sense el bit ocult)

exponent = 1 + 127 = 128 = 10000000

signe = 1

```
z = 1 10000000 11110100000000000000110
  = 1100 0000 0111 1010 0000 0000 0000 0110
  = 0xC07A0006
```