



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Estructura de Computadores

Tema 5: Aritmética de Enteros y Coma Flotante

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



Introducción

Este tema tiene por objetivo responder a las siguientes preguntas:

- ❑ ¿Qué pasa si hacemos una suma (resta) y el resultado no cabe en el registro resultado?
 - Y ¿cuándo ocurre?
- ❑ ¿Cómo es el hardware que utilizamos para multiplicar y dividir?
- ❑ ¿Cómo representamos números fraccionarios o números reales?
 - ¿Cómo operamos con ellos?

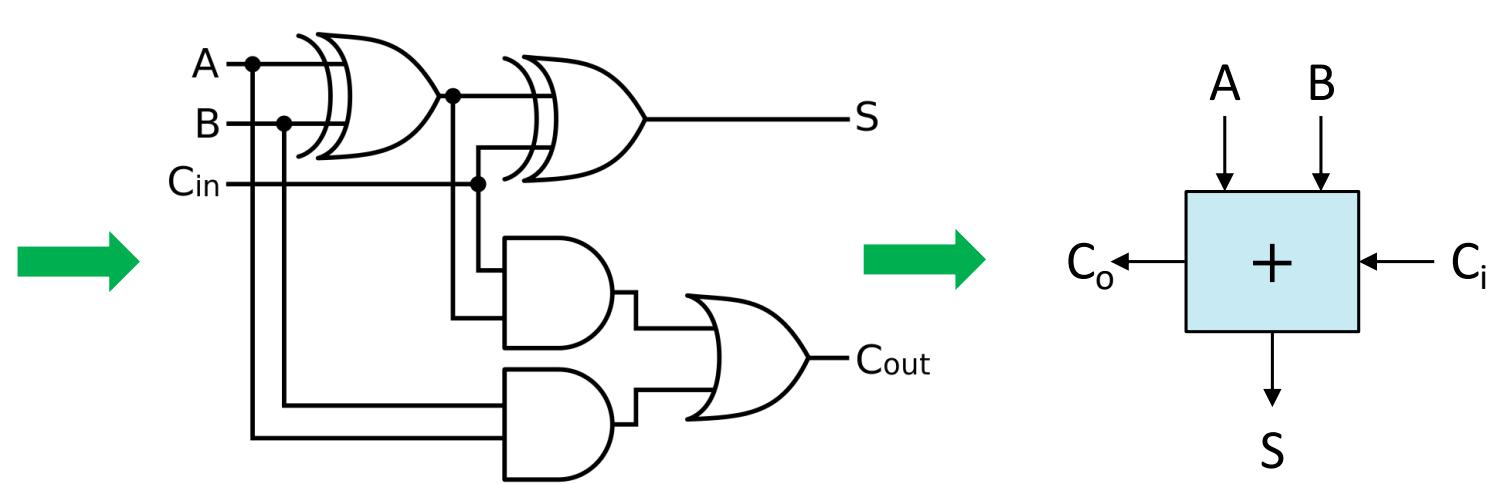
Aritmética de Enteros

- ❑ Overflow de suma y resta de enteros (y naturales)
- ❑ Multiplicación de enteros de 32 bits con resultado de 64 bits
- ❑ División de enteros de 32 bits con cálculo del resto

Suma y Resta

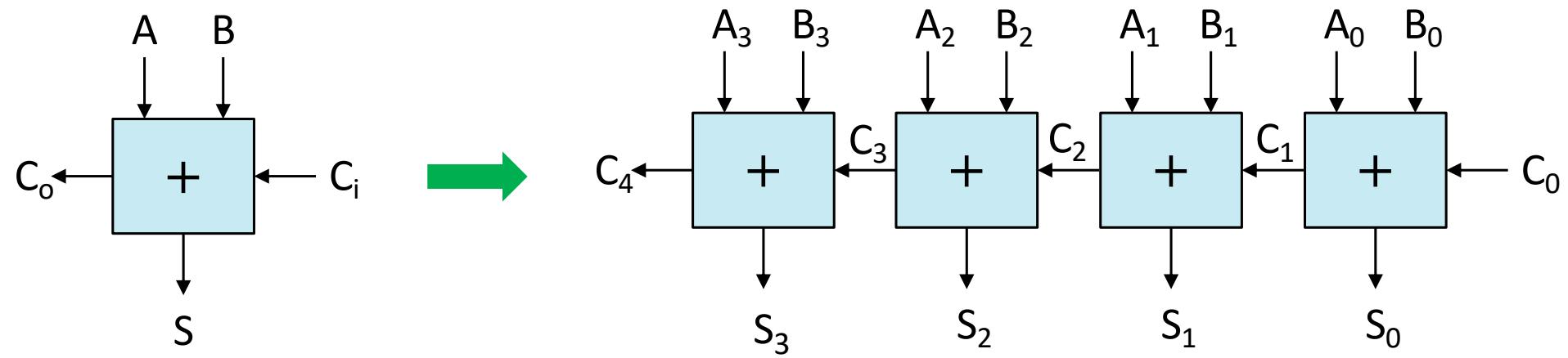
- La suma de números enteros en ca2 usa el mismo hardware que la suma de números naturales.

A	B	C _i	C _o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Suma y Resta

- La suma de números enteros en ca2 usa el mismo hardware que la suma de números naturales.



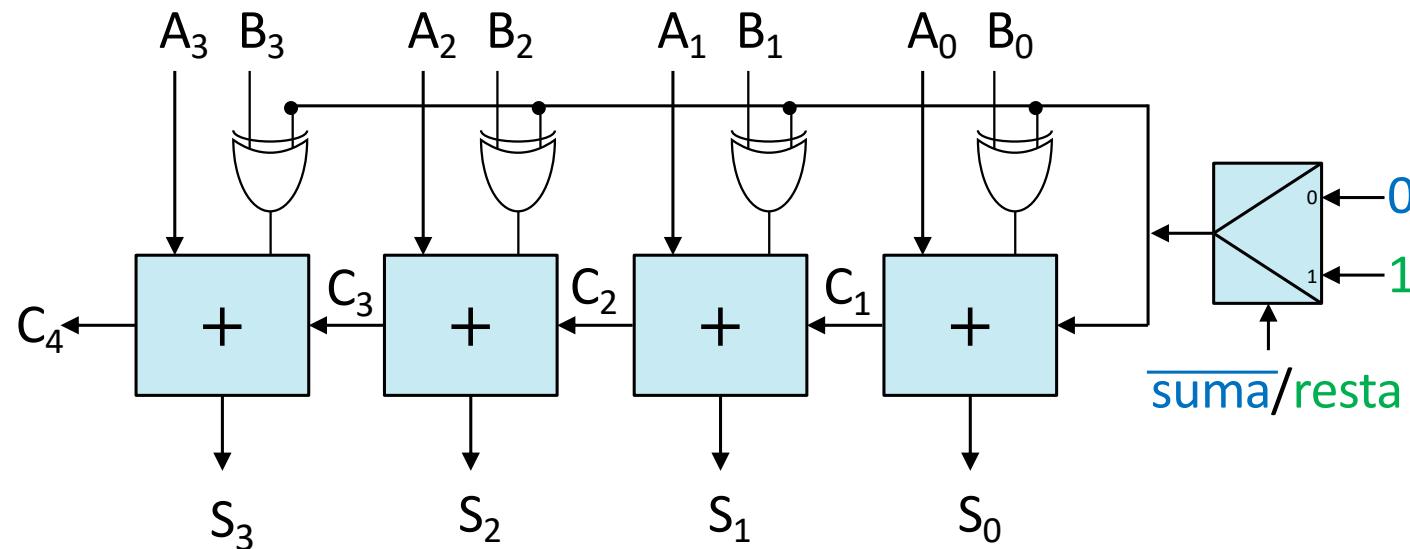
Sumador de 4 bits

Suma y Resta

- Para restar dos números enteros (o naturales) podemos hacer lo siguiente:

$$A - B = A + (-B) = A + (\sim B + 1)$$

- Entonces, podremos usar el mismo sumador (+ hardware adicional) para las restas:



Sumador/Restador de 4 bits

Overflow de la Suma y Resta

- Queremos saber cómo detectar que el resultado de una operación de +/- con números enteros en ca2 de n bits no es representable en n bits, se ha producido un **overflow**.
- Rango de un número entero en ca2 de n bits: $-2^{n-1} \dots 2^{n-1}-1$
- La forma más simple de detectar el overflow de la suma es:
 - 2 Números enteros en ca2 de diferente signo nunca pueden provocar overflow.

$$c = a+b, \text{ con } a \leq 0 \text{ y } b \geq 0 \Rightarrow a \leq a+b \leq b$$

- Sólo puede haber overflow si los dos números tienen el mismo signo

Se produce overflow en $c = a+b$, cuando a y b son del mismo signo y c de signo contrario.

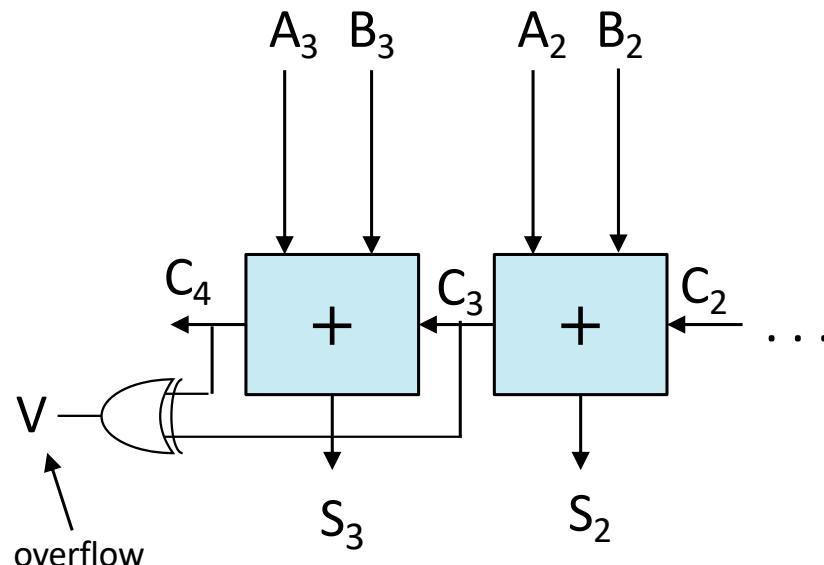
$c = a+b$,
 $C_{n-1} = A_{n-1} = B_{n-1}$
Es correcto si $C_{n-1} = C_n$

El overflow se implementa con $C_n \oplus C_{n-1} = C_n \wedge C_{n-1}$

Overflow de la Suma y Resta

- ¿Overflow de la suma?

$$\text{overflow} = C_n \oplus C_{n-1}$$



correcto

overflow

Sin posibilidad
de overflow
 $A_3 \neq B_3$

overflow

correcto

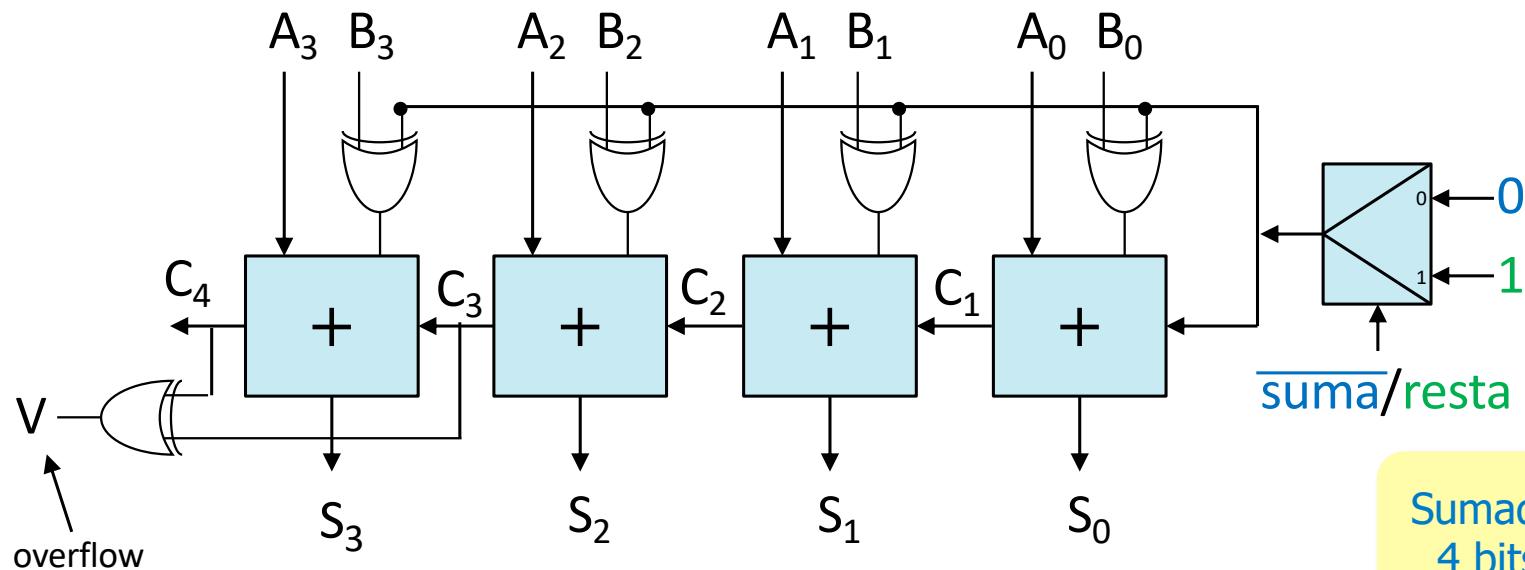
A_3	B_3	C_3	C_4	S_3	
0	0	0	0	0	$C_3 == C_4$
0	0	1	0	1	$C_3 \neq C_4$
0	1	0	0	1	
0	1	1	1	0	
1	0	0	0	1	$C_3 == C_4$
1	0	1	1	0	$C_3 \neq C_4$
1	1	0	1	0	
1	1	1	1	1	$C_3 == C_4$

Overflow de la Suma y Resta

- ¿Overflow de la resta?

$$d = a - b \Rightarrow a = d + b$$

Se produce overflow en la resta si el resultado (d) y el sustraendo (b) son del mismo signo y el minuendo (a) de signo contrario.



Sumador/Restador de 4 bits con overflow

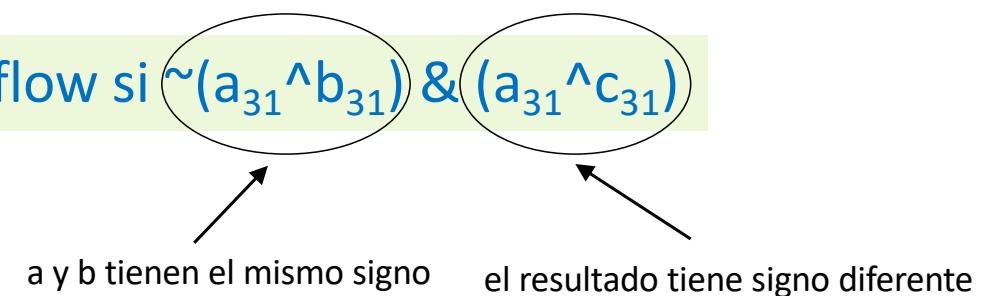
Overflow de la Suma y Resta

- ❑ Algunos Lenguajes de Alto Nivel requieren que el programa genere una excepción si se produce un overflow.
- ❑ MIPS implementa por hardware esta detección.

Instrucciones	Acción al detectar overflow
add, addi, sub	Genera una excepción
addu, addiu, subu	Ignora

- ❑ En C, el overflow se ignora.
 - Si necesitamos calcularlo, hay que hacerlo por programa

si $c = a + b$ son int, hay overflow si $\sim(a_{31} \wedge b_{31}) \& (a_{31} \wedge c_{31})$



Detección de Overflow en MIPS

- ❑ MIPS no incluye instrucciones específicas para calcular si se ha producido overflow.
- ❑ Hay que calcularlo con $V = \sim(a_{31} \wedge b_{31}) \wedge (a_{31} \wedge c_{31})$
- ❑ Detección del overflow en MIPS:

```
addiu $t2,$t1,$t0      # c = a+b  
  
xor $t3,$t1,$t0          # a^b  
nor $t3,$t3,$zero        # ~(a^b)  
xor $t4,$t2,$t0          # a^c  
and $t3,$t3,$t4          # ~(a^b) & a^c  
srl $t3,$t3,31           # Ponemos V en el bit 0
```

Overflow de la Suma y Resta de Naturales

- ❑ Queremos saber cómo detectar que el resultado de una operación de +/- con números naturales n bits no es representable en n bits, se ha producido un **overflow**.
- ❑ Rango de un número natural de n bits: $0 .. 2^{n-1}$
- ❑ En hardware es muy simple, es el bit de acarreo del último sumador:
 - Lo llamamos **carry** para la suma
 - Lo llamamos **borrow** para la resta
- ❑ Se puede detectar por software, sabiendo que:
 - $c = a + b$, se produce carry (overflow) si $c < a$ (naturales) [$b = c - a$]
 - $c = a - b$, se produce borrow (overflow) si $a < b$ (naturales)

```
addiu $t2,$t0,$t1    # c = a+b  
  
sltu $t3,$t2,$t0    # $t3=carry
```

Ejemplo práctico de uso del carry

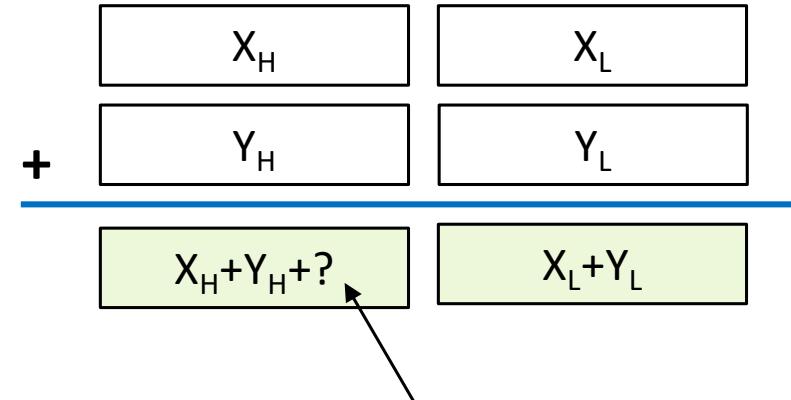
- Dada la siguiente declaración en C:

```
long long x, y; // Números enteros de 8 bytes
```

PROBLEMA 5.9

- Calculad en MIPS: **x + y**

```
la $t6,y  
lw $t0,0($t6)      # $t0 = YL  
lw $t1,4($t6)      # $t1 = YH  
  
la $t6,x  
lw $t2,0($t6)      # $t2 = XL  
lw $t3,4($t6)      # $t3 = XH  
addu $t2,$t0,$t2    # XL+YL  
slt $t5,$t2,$t0     # $t5 = carry  
addu $t3,$t1,$t3    # XH+YH  
addu $t3,$t3,$t5     # XH+YH+ carry  
sw $t2,0($t6)       # XL = $t2  
sw $t3,4($t6)       # XH = $t3
```



Para que funcione hay que sumar el carry de X_L+Y_L.

c = a + b, carry = c < a

Ejemplo práctico de uso del borrow

- Dada la siguiente declaración en C:

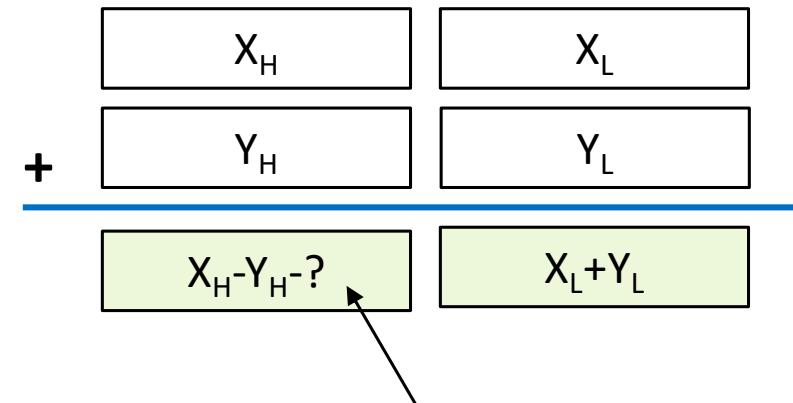
```
long long x, y; // Números enteros de 8 bytes
```

PROBLEMA 5.9

- Calculad en MIPS: $x - y$

```
# $t1 = YH : $t0 = YL
# $t3 = XH : $t2 = XL

subu $t4,$t2,$t0    # XL-YL
sltu $t6,$t2,$t0    # $t6 = borrow
subu $t5,$t3,$t1    # XH+YH
subu $t5,$t5,$t6    # XH-YH- borrow
la $t6,x
sw $t4,0($t6)       # XL = $t4
sw $t5,4($t6)       # XH = $t5
```



Para que funcione hay que restar el borrow de $X_H - Y_H$.

$c = a - b$, $\text{borrow} = a < b$

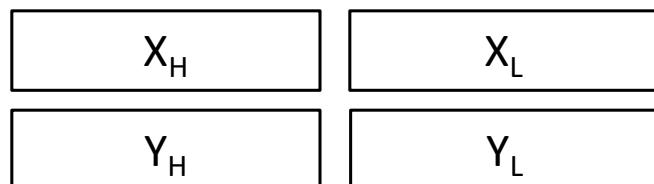
Ejemplo práctico

- Dada la siguiente declaración en C:

```
long long x, y; // Números enteros de 8 bytes
```

PROBLEMA 5.9

- Traducid a MIPS: **if (x > y) x = 1;**



$(x > y)$, 3 casos:

- $X_H > Y_H \rightarrow$ cierto
- $X_H < Y_H \rightarrow$ falso
- $X_H == Y_H$
 - $X_L > Y_L \rightarrow$ cierto
 - $X_L \leq Y_L \rightarrow$ falso

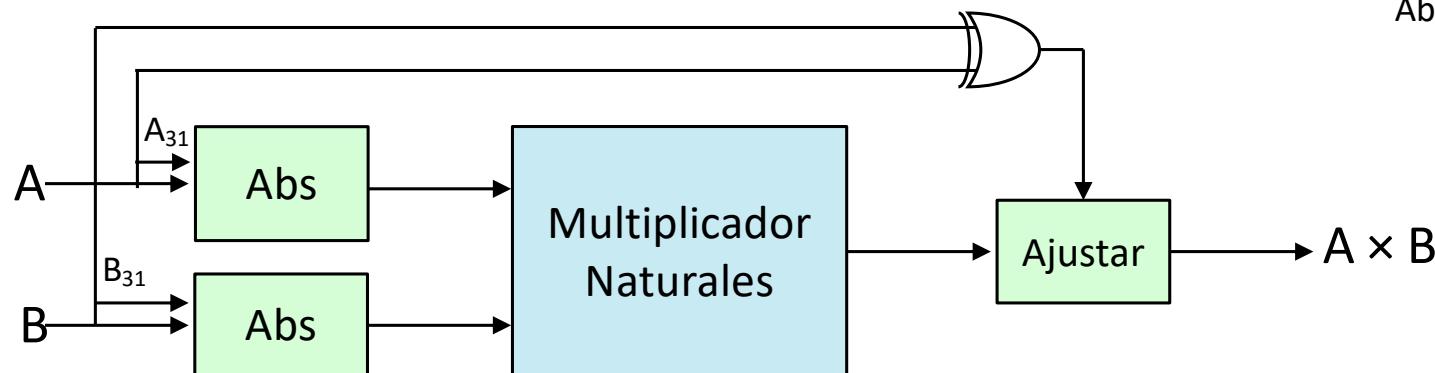
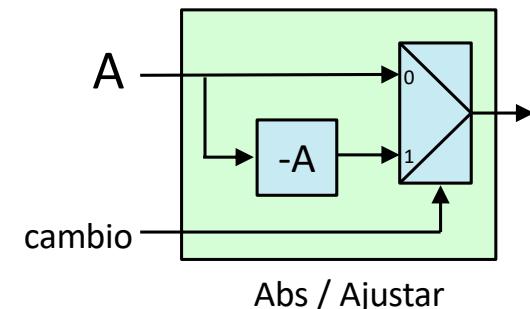
- También se puede traducir haciendo la resta:

```
if (x-y > 0) x=1;
```

Esta comparación
es unsigned

Multiplicación de enteros de 32b con resultado en 64b

- Para multiplicar enteros usaremos el mismo algoritmo que utilizamos manualmente:
 - Calcularemos los valores absolutos.
 - Haremos el producto de naturales.
 - Cambiaremos el signo del resultado si es necesario.



Algoritmo Tradicional de Multiplicación de naturales

- El algoritmo que aprendimos en la escuela

$$\begin{array}{r} 348 \\ \times 951 \\ \hline 348 \\ 1740 \\ + 3132 \\ \hline 330948 \end{array}$$

← Multiplicando
← Multiplicador
← Resultado

$$\begin{array}{r} 348 \\ \times 951 \\ \hline 348 \\ 17400 \\ + 313200 \\ \hline 330948 \end{array}$$

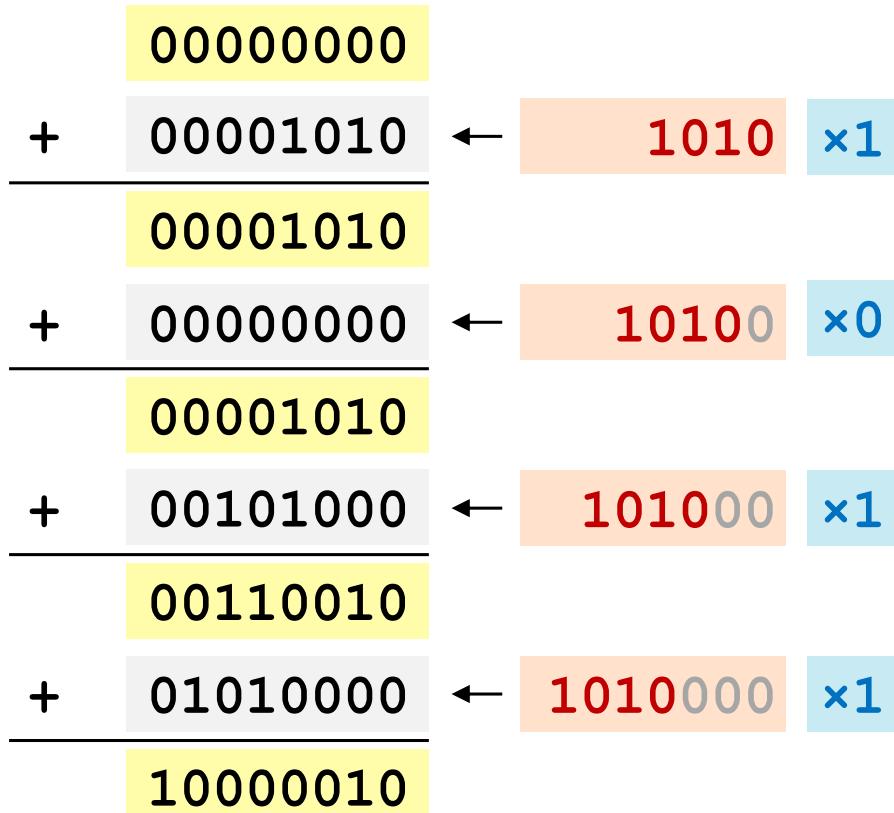
Algoritmo Tradicional de Multiplicación de naturales

- En binario es más simple porque sólo podemos multiplicar por 0 o por 1.

$$\begin{array}{r} \textcolor{red}{1010} \leftarrow \text{Multiplicando (10)} \\ \times \textcolor{blue}{1101} \leftarrow \text{Multiplicador (13)} \\ \hline 1010 & = & \textcolor{red}{1010} & \times & \textcolor{blue}{1} \\ 0000 & = & \textcolor{red}{10100} & \times & \textcolor{blue}{0} \\ 1010 & = & \textcolor{red}{101000} & \times & \textcolor{blue}{1} \\ + 1010 & = & \textcolor{red}{1010000} & \times & \textcolor{blue}{1} \\ \hline 10000010 & \leftarrow & \text{Resultado (130)} \end{array}$$

Algoritmo Secuencial de Multiplicación de naturales

- Si usamos sumadores convencionales, hemos de hacer la multiplicación en serie.



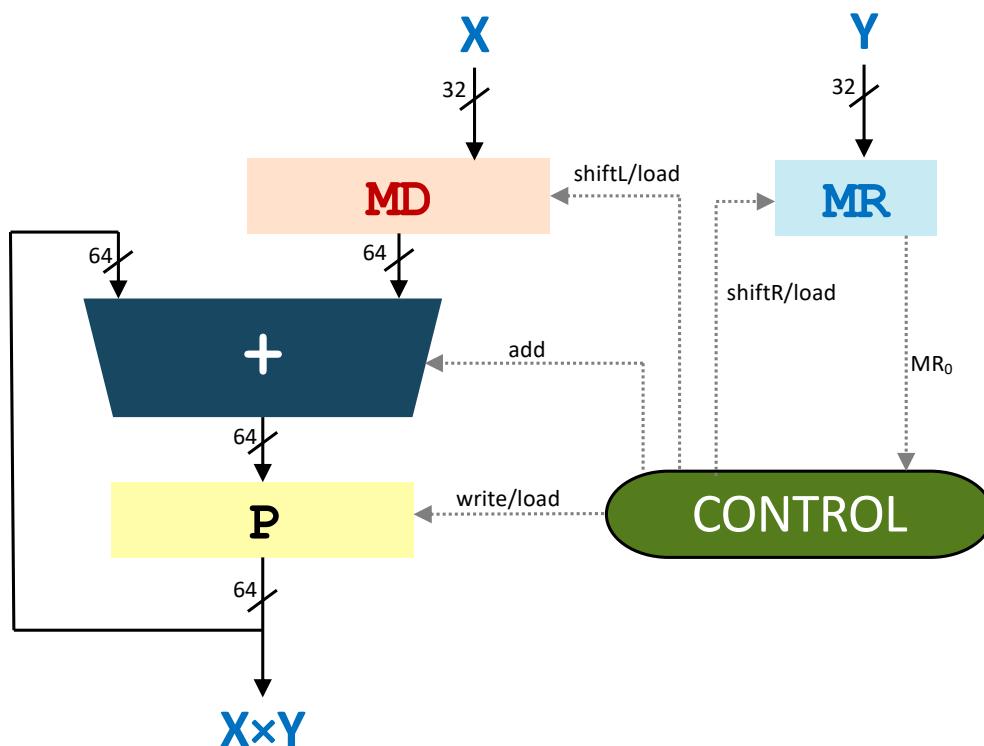
A separate binary multiplication diagram shows 1010 multiplied by 1101. The result is 10000010.

$$\begin{array}{r} 1010 \\ \times 1101 \\ \hline 1010 \\ 0000 \\ 101000 \\ + 101000 \\ \hline 10000010 \end{array}$$

```
P = 0;  
MD = X;  
MR = Y;  
for (i=0; i<n; i++) {  
    if (MR0 == 1)  
        P = P+MD;  
    MD = MD << 1;  
    MR = MR >> 1;  
}
```

Circuito para la Multiplicación Secuencial de naturales

- ❑ Naturales de 32 bits con resultado de 64 bits: MD y P de 64 bits, MR de 32 bits.
 - Si la suma tarda 1 ciclo, el producto tardará 33 ciclos.



```
MD63:32 = 0; MD31:0 = X;  
MR = Y;  
P = 0;  
for (i=1; i<=32; i++) {  
    if (MR0 == 1)  
        P = P+MD;  
    MD = MD << 1;  
    MR = MR >> 1;  
}
```

Ejemplo de Multiplicación Secuencial de naturales

□ X = 1010, Y=1101 de 4 bits, resultado en P de 8 bits

- Inicio: Inicializamos $MD_{0:3} = X$, $MR = Y$, $P = 0$

Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	

```
MD63:32 = 0; MD31:0 = X;  
MR = Y;  
P = 0;  
for (i=1; i<=32; i++) {  
    if (MR0 == 1)  
        P = P+MD;  
    MD = MD << 1;  
    MR = MR >> 1;  
}
```

Ejemplo de Multiplicación Secuencial de naturales

- iter 1, como $MR_0 == 1$, sumamos P+D

$$\begin{aligned}P &= 00000000 \\MD &= 00001010 \\P+MD &= 00001010\end{aligned}$$

Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	
1													0	0	0	0	1	0	1	0

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

Ejemplo de Multiplicación Secuencial de naturales

- iter 1, como $MR_0 == 1$, sumamos P+D, desplazamos MD a la izquierda y MR a la derecha

$$\begin{array}{ll} P & = 00000000 \\ MD & = 00001010 \\ P+MD & = 00001010 \end{array}$$

Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	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

← →

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

Ejemplo de Multiplicación Secuencial de naturales

- iter 2, como $MR_0 == 2$, P no se modifica,

Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	
1	0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	1	0	1	0	
2													0	0	0	0	1	0	1	0

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

Ejemplo de Multiplicación Secuencial de naturales

- iter 2, como $MR_0 == 2$, P no se modifica,
desplazamos MD a la izquierda y MR a la derecha

Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	
1	0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	1	0	1	0	
2	0	0	1	0	1	0	0	0	0	1	1	0	0	0	0	1	0	1	0	



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

Ejemplo de Multiplicación Secuencial de naturales

- iter 3, como $MR_0 == 1$, sumamos P+D,

$$\begin{aligned}
 P &= 00001010 \\
 MD &= 00101000 \\
 P+MD &= 00110010
 \end{aligned}$$

Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	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	
2	0	0	1	0	1	0	0	0	0	1	1	1	0	0	0	0	1	0	1	
3													0	0	1	1	0	0	1	

```

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

```

Ejemplo de Multiplicación Secuencial de naturales

- iter 3, como $MR_0 == 1$, sumamos P+D, desplazamos MD a la izquierda y MR a la derecha

Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	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	
2	0	0	1	0	1	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	



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

Ejemplo de Multiplicación Secuencial de naturales

□ iter 4, como $MR_0 == 1$, sumamos P+D,

$$\begin{aligned} P &= 00110010 \\ MD &= 01010000 \\ P+MD &= 10000010 \end{aligned}$$

Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	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	
2	0	0	1	0	1	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	0	1	0	1	1	0	0	1	
4													1	0	0	0	0	0	1	

```
if (MR0 == 1)
    P = P+MD;
```

Ejemplo de Multiplicación Secuencial de naturales

- iter 4, como $MR_0 == 1$, sumamos P+D, desplazamos MD a la izquierda y MR a la derecha

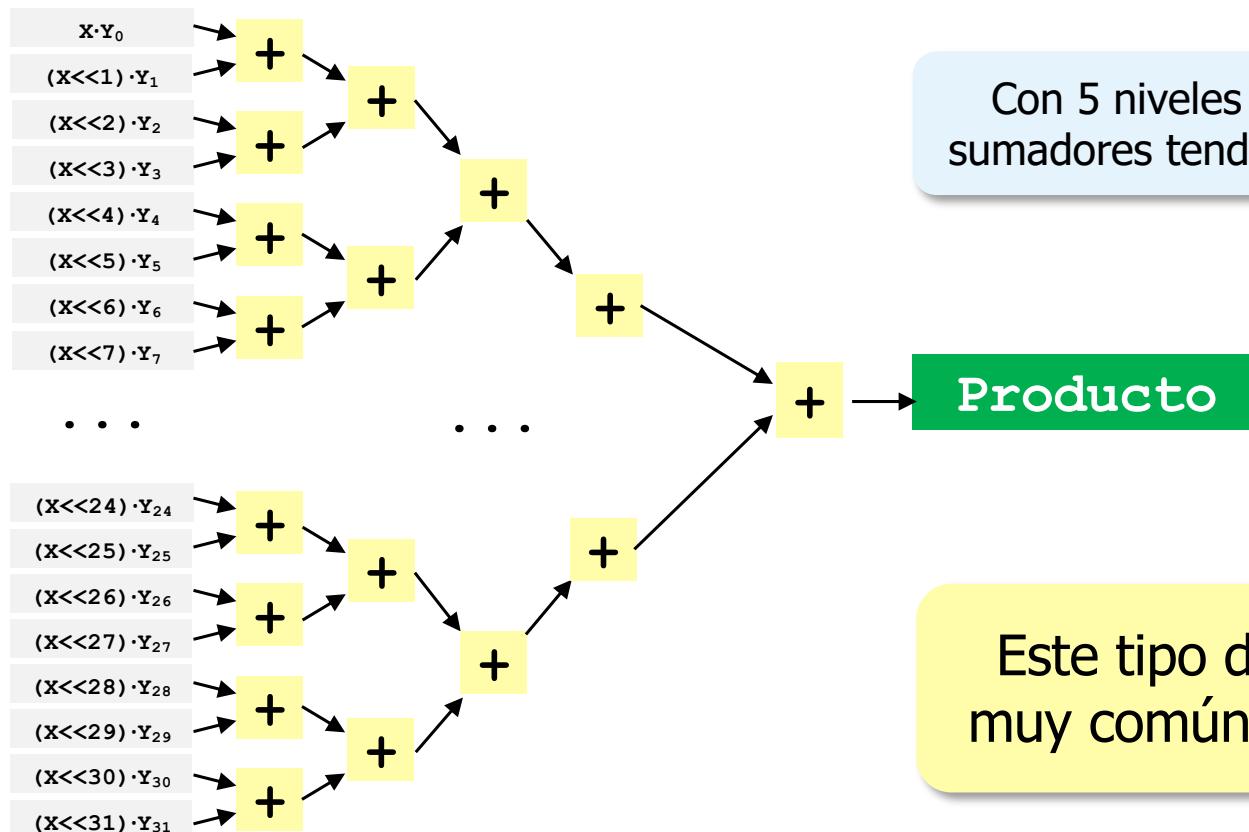
Iteración	MD								MR				P							
Inicial	0	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	
1	0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	0	1	0	1	0
2	0	0	1	0	1	0	0	0	0	1	1	0	0	0	0	1	0	1	0	
3	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1	0
4	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	



$MD = MD \ll 1;$
 $MR = MR \gg 1;$

Multiplicadores Rápidos

- El multiplicador secuencial es muy lento. Existen implementaciones hardware mucho más eficientes.



Con 5 niveles ($\log_2 32 = 5$) de sumadores tendremos el producto.

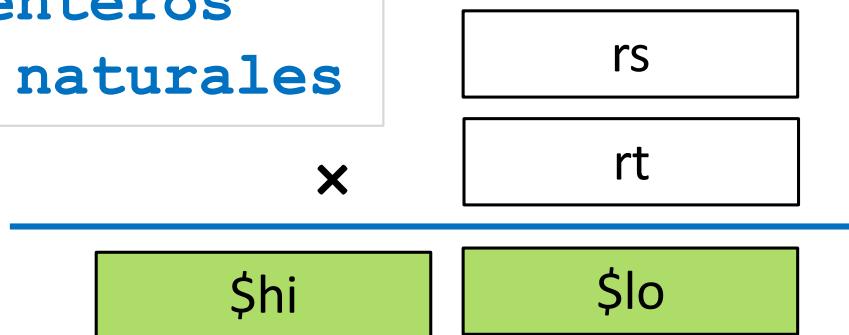
Producto

Este tipo de algoritmo es muy común: **REDUCCIÓN**

Instrucciones para Multiplicar en MIPS

- Instrucción en MIPS para multiplicar 2 números de 32 bits

```
mult rs, rt # $hi:$lo = rs*rt, enteros  
multu rs, rt # $hi:$lo = rs*rt, naturales
```



- El resultado (64 bits) se guarda en los registros **\$hi** y **\$lo**
 - Son registros especiales. No se pueden usar en las instrucciones que hemos visto
 - Para acceder a estos registros necesitamos instrucciones especiales

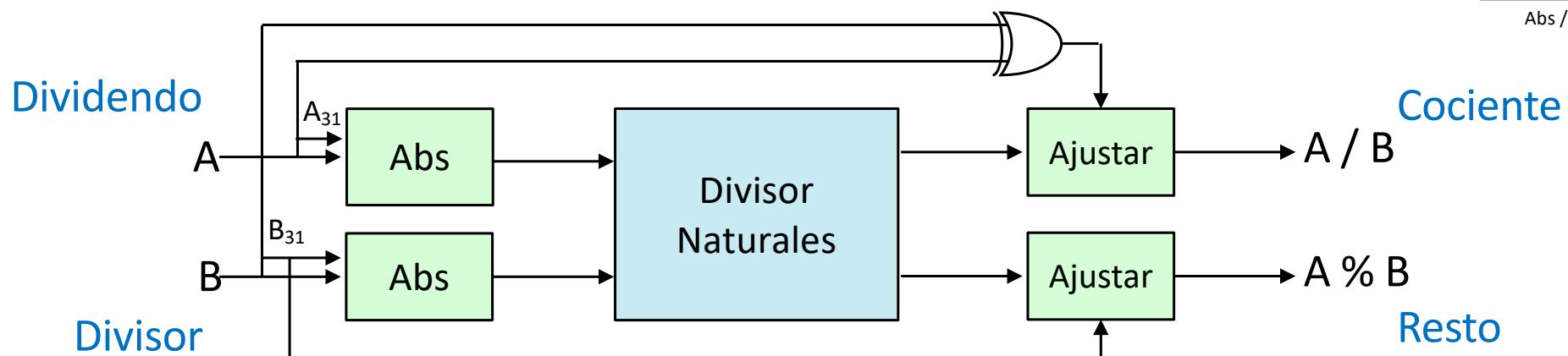
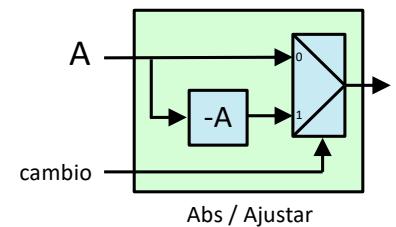
```
mflo rd      # rd = $lo  
mfhi rd      # rd = $hi
```

Overflow en MIPS al Multiplicar

- ❑ En C y otros Lenguajes de Programación, el resultado de multiplicar 2 números de 32 bits se guarda en 32 bits. Los 32 bits altos se ignoran.
- ❑ Si queremos saber si se ha producido overflow en MIPS, hemos de comprobar manualmente las siguientes condiciones:
 - **multu** tiene overflow si **\$hi** $\neq 0$
 - **mult** tiene overflow si **\$hi** no es la extensión de signo de **\$lo**
 - ✓ Si bit 31 de **\$lo** es 1, **\$hi** ha de ser -1, sino overflow
 - ✓ Si bit 31 de **\$lo** es 0, **\$hi** ha de ser 0, sino overflow

División de enteros

- Para dividir enteros usaremos el mismo algoritmo que utilizamos manualmente:
 - Calcularemos los valores absolutos.
 - Haremos la división de naturales, calculando el cociente y el residuo (resto).
 - Cambiaremos de signo:
 - ✓ El cociente, si los operandos son de signo diferentes
 - ✓ El resto, si el dividendo es negativo



División de enteros

❑ El signo del resto requiere una explicación.

❑ Siempre se ha de cumplir:

$$\text{Dividendo} = \text{Cociente} \times \text{Divisor} + \text{Resto}$$

❑ Para comprender cómo se determina el signo del resto:

Cociente negativo si
Dividendo y Divisor tienen
signos contrarios.

Dividendo	Divisor	Cociente	Resto	Comprobación
7	2	3	1	$7 = 3 \times 2 + 1$
-7	2	-3	-1	$-7 = (-3) \times 2 + (-1)$
7	-2	-3	1	$7 = (-3) \times (-2) + 1$
-7	-2	3	-1	$-7 = 3 \times (-2) + (-1)$

❑ Alguien podría hacer: $-7/2 = -4$ y $-7\%2 = 1$, cumple que $(-4 \times 2 + 1 = -7)$

○ Pero entonces: $-(x / y) \neq (-x) / y$

Resto y Dividendo han
de tener el mismo signo

Overflow / Errores en la división

- ❑ Si el divisor es 0, el resultado es indefinido. En C una división por cero es **undefined behaviour** y puede acabar en una excepción.
- ❑ La división de naturales nunca puede provocar un overflow.
- ❑ La división de enteros, sólo tiene un caso de overflow:
 - Si dividimos el número entero más pequeño por -1, el resultado es **no representable**
 - Con 32 bits: $-2^{31} / -1 = 2^{31}$, no representable en 32 bits.

Divisiones y Desplazamientos (srl y sra)

- ❑ Comentamos que es posible sustituir algunas divisiones por desplazamientos a la derecha: **srl** y **sra**.
- ❑ Podemos hacerlo cuando tengamos divisores potencia de 2 (2, 4, 8, 16, ...).
- ❑ Dividiendo naturales (con divisor 2^x), **srl** da el mismo resultado que **divu**.
- ❑ Dividiendo enteros (con divisor 2^x), **sra** , **NO** da el mismo resultado que **divu**, si el dividendo es negativo.
- ❑ En general, traduciremos siempre las divisiones y módulos (/ y %) a **div** y **divu**, dependiendo del tipo (int o unsigned)

Instrucciones para Dividir en MIPS

- Instrucción en MIPS para dividir 2 números de 32 bits

```
div rs, rt # $lo = rs/rt; $hi = rs%rt enteros  
divu rs, rt # $lo = rs/rt; $hi = rs%rt naturales
```

- Los 2 resultados se guardan en los registros **\$hi** y **\$lo**
 - Son registros especiales. No se pueden usar en las instrucciones que hemos visto
 - Para acceder a estos registros necesitamos instrucciones especiales

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

Algoritmo de División en base 10

$$421 \quad | \quad 13$$

1) 13 \rightarrow No cabe

$$\begin{array}{r} - 000 \\ \hline 421 \end{array}$$

2) 13 \rightarrow Cabe a 3, $13 \times 3 = 39$

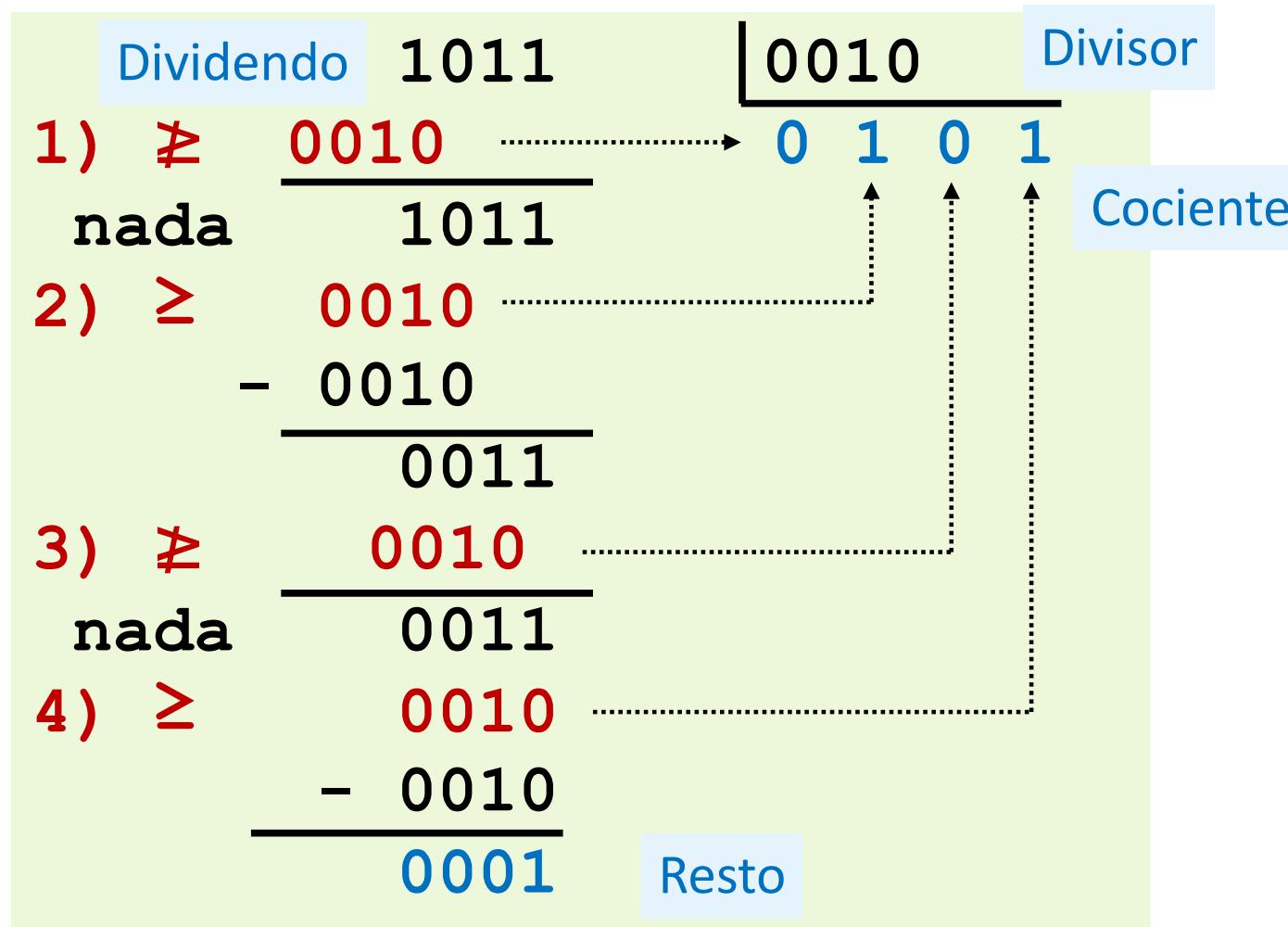
$$\begin{array}{r} - 39 \\ \hline 31 \end{array}$$

3) 13 \rightarrow Cabe a 2, $13 \times 2 = 26$

$$\begin{array}{r} - 26 \\ \hline 5 \end{array}$$

\rightarrow Resto 5, Cociente 32

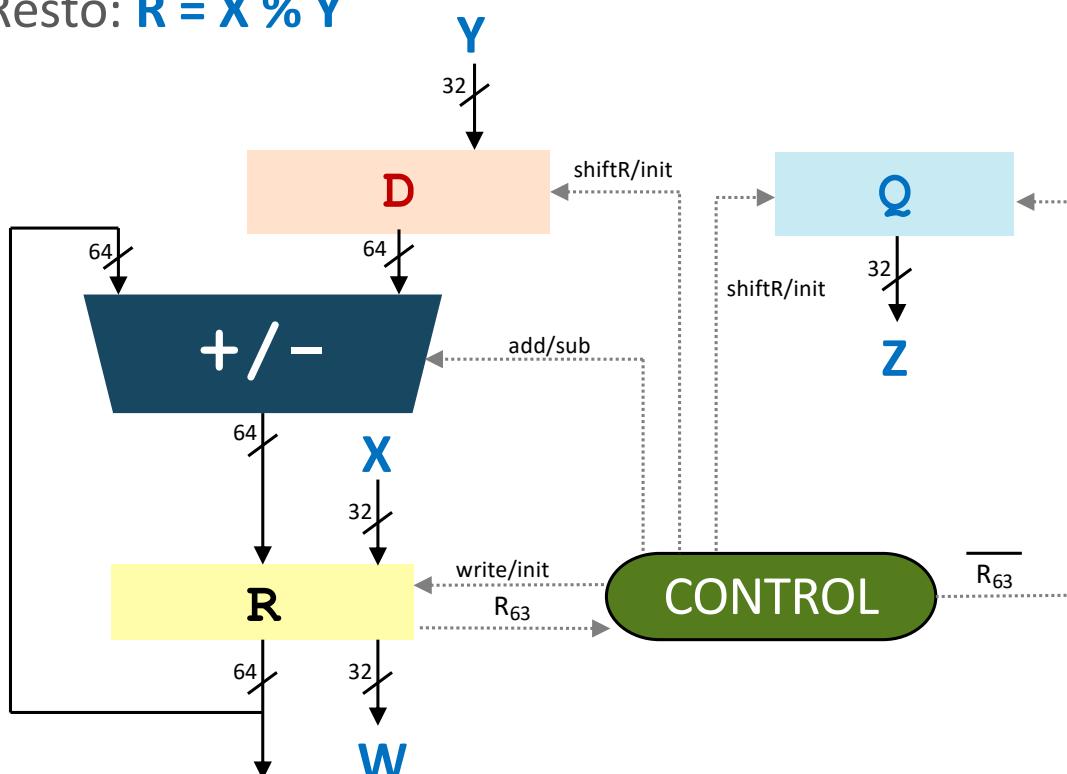
Algoritmo de División en base 2



Circuito para la División de naturales

- División de Naturales de 32 bits “con restauración”.

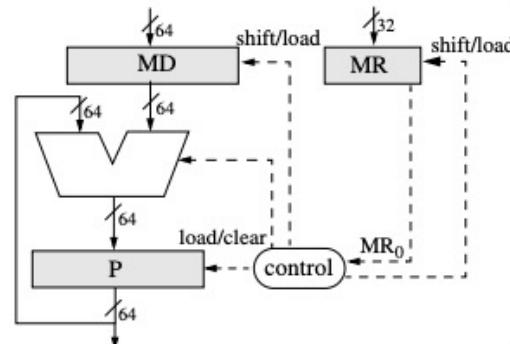
- Cociente: $Q = X / Y$
- Resto: $R = X \% Y$



```
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;  
    }  
}
```

Problema Examen Final

A continuació es mostra la unitat de procés i l'algorisme que usa la unitat de control (UC) del multiplicador seqüencial de nombres naturals de 32 bits estudiat a classe, que calcula $Z=X*Y$:



```
MR = Y;
MD63:32 = 0; MD31:0 = X;
P63:32 = 0; P31:0 = 0;
for (i=1; i<=32; i++)
{
    if (MR0 == 1)
        { P = P + MD; }
    MD = MD << 1;
    MR = MR >> 1;
}
Z = P;
```

Per implementar la UC cal considerar que té un senyal d'entrada MR_0 que correspon al bit de menys pes del registre MR i 6 sortides ($load_MD$, $load_MR$, $load_P$, $shift_MD$, $shift_MR$, $clear_P$). Els senyals *load* carreguen sincronament al registre corresponent el que tinguin a la seva entrada. El senyal $shift_MD$ permet fer el desplaçament d'un bit a l'esquerra del registre MD , el senyal $shift_MR$ permet fer el desplaçament d'un bit a la dreta del registre MR . El senyal $clear_P$ posa a 0 el registre P . Tots aquests senyals són actius amb valor 1. Amb valor 0 són inactius. De les dues tasques que pot fer un registre, si les dues estan actives, sols executarà la càrrega per ser més prioritària. Suposem que l'ALU només realitzarà sumes en tot moment i, per tant no cal especificar la funció. També considerem que el control de les iteracions ja està implementat internament dins la unitat de control.

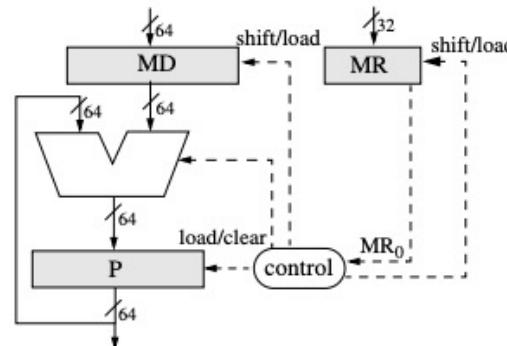
- a) Indica els valors que falten a la següent taula dels senyals de sortida durant la inicialització i durant cada iteració del processament (valen el mateix a totes les iteracions). Els valors que falten poden ser 1, 0 o X (*don't care*, és a dir, que sigui indiferent el valor que prengui).

	load_MD	load_MR	load_P	shift_MD	shift_MR	clear_P
inici						
cada iteració			MR ₀			

Examen Final junio 2019
Pregunta 6

Problema Examen Final

A continuació es mostra la unitat de procés i l'algorisme que usa la unitat de control (UC) del multiplicador seqüencial de nombres naturals de 32 bits estudiat a classe, que calcula $Z=X*Y$:



```
MR = Y;  
MD63:32 = 0; MD31:0 = X;  
P63:32 = 0; P31:0 = 0;  
for (i=1; i<=32; i++)  
{  
    if (MR0 == 1)  
        { P = P + MD; }  
    MD = MD << 1;  
    MR = MR >> 1;  
}  
Z = P;
```

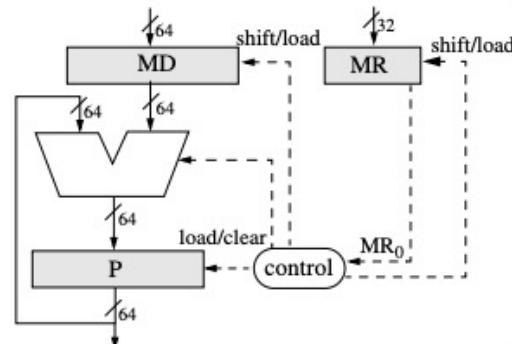
Per implementar la UC cal considerar que té un senyal d'entrada MR_0 que correspon al bit de menys pes del registre MR i 6 sortides ($load_MD$, $load_MR$, $load_P$, $shift_MD$, $shift_MR$, $clear_P$). Els senyals $load$ carreguen sincronament al registre corresponent el que tinguin a la seva entrada. El senyal $shift_MD$ permet fer el desplaçament d'un bit a l'esquerra del registre MD , el senyal $shift_MR$ permet fer el desplaçament d'un bit a la dreta del registre MR . El senyal $clear_P$ posa a 0 el registre P . Tots aquests senyals són actius amb valor 1. Amb valor 0 són inactius. De les dues tasques que pot fer un registre, si les dues estan actives, sols executarà la càrrega per ser més prioritària. Suposem que l'ALU només realitzarà sumes en tot moment i, per tant no cal especificar la funció. També considerem que el control de les iteracions ja està implementat internament dins la unitat de control.

Examen Final junio 2019
Pregunta 6

	load_MD	load_MR	load_P	shift_MD	shift_MR	clear_P
inici	1	1	0	X	X	1
cada iteració	0	0	MR ₀	1	1	0

Problema Examen Final

A continuació es mostra la unitat de procés i l'algorisme que usa la unitat de control (UC) del multiplicador seqüencial de nombres naturals de 32 bits estudiat a classe, que calcula $Z=X*Y$:



```
MR = Y;
MD63:32 = 0; MD31:0 = X;
P63:32 = 0; P31:0 = 0;
for (i=1; i<=32; i++)
{
    if (MR0 == 1)
        { P = P + MD; }
    MD = MD << 1;
    MR = MR >> 1;
}
Z = P;
```

Per implementar la UC cal considerar que té un senyal d'entrada MR_0 que correspon al bit de menys pes del registre MR i 6 sortides ($load_MD$, $load_MR$, $load_P$, $shift_MD$, $shift_MR$, $clear_P$). Els senyals *load* carreguen sincronament al registre corresponent el que tinguin a la seva entrada. El senyal *shift_MD* permet fer el desplaçament d'un bit a l'esquerra del registre MD , el senyal *shift_MR* permet fer el desplaçament d'un bit a la dreta del registre MR . El senyal *clear_P* posa a 0 el registre P . Tots aquests senyals són actius amb valor 1. Amb valor 0 són inactius. De les dues tasques que pot fer un registre, si les dues estan actives, sols executarà la càrrega per ser més prioritària. Suposem que l'ALU només realitza sumes en tot moment i, per tant no cal especificar la funció. També considerem que el control de les iteracions ja està implementat internament dins la unitat de control.

- b) Volem fer una optimització de la UC de forma que el bucle de processament acabi quan el registre MR no tingui cap bit a 1. Indica els canvis que s'haurien de realitzar a l'algorisme donat per reflectir aquesta optimització.

Examen Final junio 2019
Pregunta 6

```
...
for (i=1; i<=32; i++) {
    ...
}
```



```
...
while (MR > 0) {
    ...
}
```

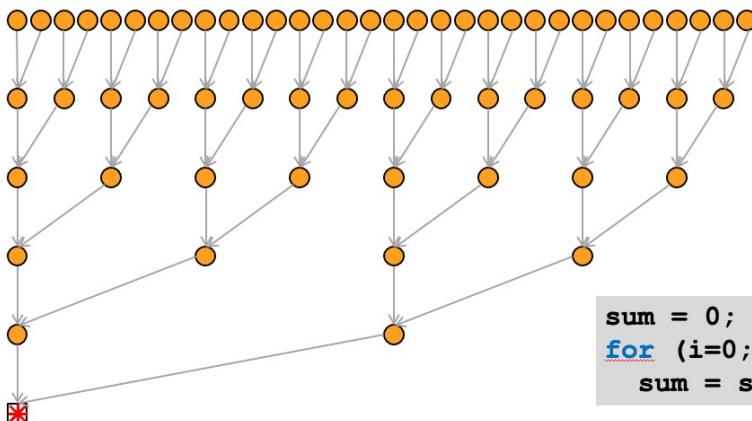
Aritmética de Coma Flotante

- ❑ Introducción
- ❑ Estándar IEEE-754
- ❑ Rango, Precisión
- ❑ Codificaciones especiales, underflow y números denormales
- ❑ Redondeo
- ❑ Conversiones entre base 10 y base 2
- ❑ Operaciones: suma, resta, bits de guarda, multiplicación y división
- ❑ Coma Flotante en MIPS
- ❑ Asociatividad de la suma

Antecedentes

- Los números en coma flotante son “peligrosos”

Kernel 01



Algún comentario sobre CF (float vs double)

Esta práctica estaba diseñada para trabajar con float.

Resultado GPU: 8389059.0
Resultado CPU: 8389376.0
Diferencia : 317.0
Error : 0.00004

Resultado GPU: 8388595.5
Resultado CPU: 8388490.0
Diferencia : 105.5
Error : 0.00001

Resultado GPU: 8387599.5
Resultado CPU: 8386772.0
Diferencia : 827.5
Error : 0.00010

Kernel01 con float

Resultado GPU: 8388336.1
Resultado CPU: 8388336.1
Diferencia : 0.0
Error : 0.00000

Resultado GPU: 8390269.9
Resultado CPU: 8390269.9
Diferencia : 0.0
Error : 0.00000

Resultado GPU: 8388889.0
Resultado CPU: 8388889.0
Diferencia : 0.0
Error : 0.00000

Kernel01 con double

Punto Fijo

- ¿Cómo representamos números reales y/o fraccionarios?
 - Imprescindibles para la física, la ingeniería, ...
- Primera idea: En **PUNTO FIJO**
 - Unos cuantos bits para la parte entera y otros para la parte fraccionaria
 - Ejemplo con 8 bits:

eeeeefff → parte entera y parte fraccionaria

10101.110 =

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

$$= 16 + 4 + 1 + 0,5 + 0,25 = 21.75$$

- El rango es bastante limitado
- La distancia entre los números representables es equidistante.

Coma Flotante

□ En **COMA FLOTANTE** (base 10)

- También conocida como notación científica o exponencial

$$v = \pm m \times 10^e \rightarrow m = \text{mantisa}, e = \text{exponente}$$

- Rango mucho más grande que en punto fijo.
- Pero la distancia entre los números representables no es equidistante.

□ Notación científica NORMALIZADA (base 10)

$$v = \pm m \times 10^e \rightarrow \text{tal que } 1.\text{xxx} \leq m \leq 9.\text{xxx}$$

- La parte entera ha de tener 1 solo dígito, entre 1 y 9.
- Ejemplos:
 - ✓ $2.34 \times 10^6 \rightarrow \text{NORMALIZADO}$
 - ✓ $0.234 \times 10^7 \rightarrow \text{NO NORMALIZADO}$

Coma Flotante en base 2

$$v = \pm 1.\text{ffff...f} \times 2^{\text{ee...e}}$$

parte entera fracción(F) exponente (E)

signo (S) mantisa

$$v = (-1)^s \times (1+0.F) \times 2^e$$

Un forma más compacta.

- Formato: Signo, Exponente, Fracción

S EEEEEEEE FFFFFFFFFFFFFFFFFFFF

- Signo:** 0 = positivo, 1 = negativo
- Exponente:** entero representado “en exceso”
- Mantisa:** NORMALIZADA
 - ✓ La parte entera siempre vale 1, es implícita y no se representa (bit oculto)
 - ✓ Sólo se codifica la fracción F

Coma Flotante en base 2

$$v = \pm 1.\text{ffff...f} \times 2^{\text{ee...e}}$$

parte entera fracción(F) exponente (E)

signo (S) mantisa

$$v = (-1)^s \times (1+0.F) \times 2^e$$

Un forma más compacta.

- El formato es un compromiso

S EEEEEEEE FFFFFFFFFFFFFFFFFFFF

- Si dedicamos más bits a **exponente**, tendremos **MAYOR RANGO**
- Si dedicamos más bits a **fracción**, tendremos **MAYOR PRECISIÓN**

Estándar IEEE-754

- ❑ Desde los primeros tiempos de la Informática se utilizó el formato en coma flotante.
- ❑ Pero cada fabricante usaba su propia codificación.
- ❑ La necesidad de intercambiar datos entre diferentes sistemas generó la necesidad de definir un estándar.
- ❑ IEEE-754, definido en 1985, es el estándar de Coma Flotante.
 - Se han realizado algunas actualizaciones, la más importante en 2008, la última en 2019.
- ❑ El estándar regula
 - La representación.
 - Las operaciones.
 - Los redondeos.
 - Las excepciones.

Estándar IEEE-754

- Simple Precisión (32b)



- Doble Precisión (64b)



- En C equivale a:

```
float x; // simple precisión 4 bytes (32 bits)
double y; // simple precisión 8 bytes (64 bits)
```

Hay más formatos, por ejemplo FP16

Estándar IEEE-754

- Simple Precisión (32b)



- Doble Precisión (64b)



- **Signo:** 1 bit (0, positivo; 1, negativo)
- **Exponente:** 8 bits / 11 bits
 - Codificado en exceso a 127 / 1023
 - Permite comparar magnitudes con un comparador de naturales
- **Fracción:** 23 bits / 52 bits
 - Parte fraccionaria de la mantisa
- **Parte entera = 1**
 - Bit oculto implícito, no se representa

Representación de Números Enteros en exceso a $2^{n-1}-1$

□ Representación de enteros en **exceso a $2^{n-1}-1$** :

○ Función de interpretación:

$$X = X_u - (2^{n-1} - 1)$$

○ Función de representación:

$$X_u = X + (2^{n-1} - 1)$$

Los valores están “ordenados”

El número más pequeño (-15) corresponde con el 0...000, y el número más grande (16) corresponde con el 1...111.

Exceso a 15

$X_4X_3X_2X_1X_0$	x_u	x	$X_4X_3X_2X_1X_0$	x_u	x
00000	0	-15	10000	16	1
00001	1	-14	10001	17	2
00010	2	-13	10010	18	3
00011	3	-12	10011	19	4
00100	4	-11	10100	20	5
00101	5	-10	10101	21	6
00110	6	-9	10110	22	7
00111	7	-8	10111	23	8
01000	8	-7	11000	24	9
01001	9	-6	11001	25	10
01010	10	-5	11010	26	11
01011	11	-4	11011	27	12
01100	12	-3	11100	28	13
01101	13	-2	11101	29	14
01110	14	-1	11110	30	15
01111	15	0	11111	31	16

Representación de Números Enteros en exceso a $2^{n-1}-1$

- El rango de representación de enteros en **exceso a $2^{n-1}-1$** es

$$X \in [-(2^{n-1}-1) .. 2^{n-1}]$$

- Hay un valor más en positivo que en negativo

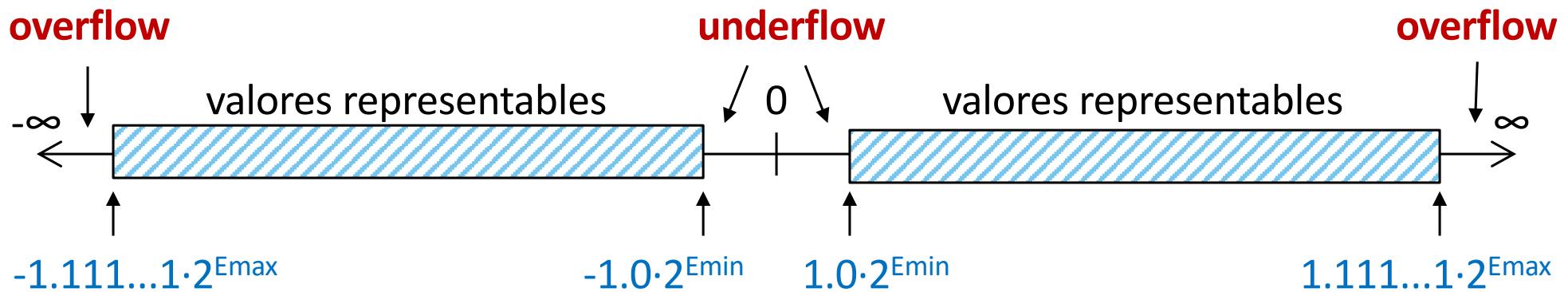
n		E_{\min}	E_{\max}
8	exceso a 127	-127	128
11	exceso a 1023	-1.023	1.024

IEEE-754, valores representables

- ❑ Mantisa $1.000\dots0 \leq \text{mantisa} \leq 1.111\dots1$

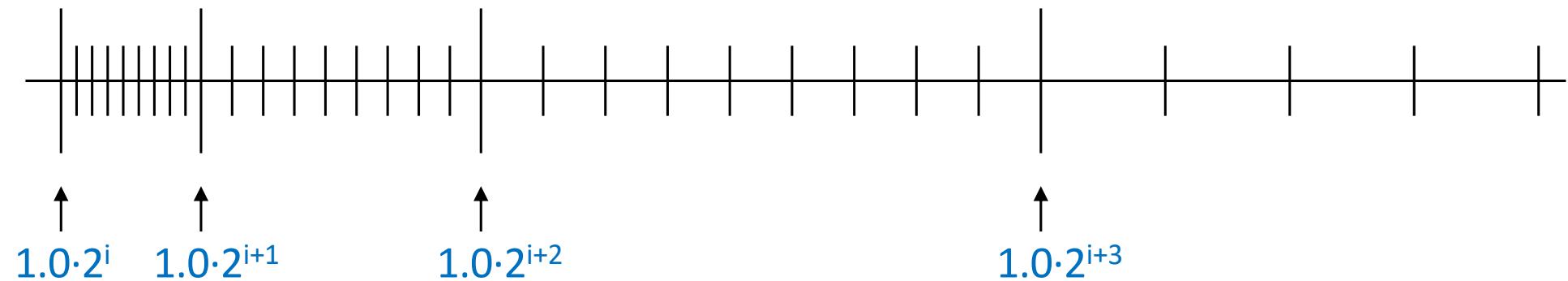
- ❑ Exponente $E_{\min} \leq E \leq E_{\max}$

- ❑ Resultados fuera de rango (**overflow**): $E > E_{\max}$
- ❑ Resultado “poco fiables” (**underflow**): magnitud $< 1.0 \cdot 2^{E_{\min}}$



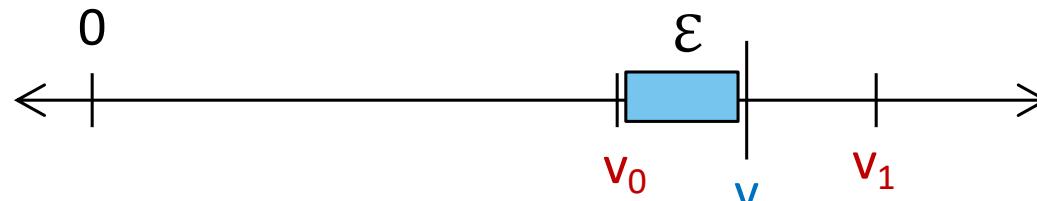
IEEE-754, valores representables

- ❑ Los valores representables NO son equidistantes.
- ❑ Con 32 bits, se pueden representar 2^{32} números diferentes, igual que con enteros, pero las distancias entre dos valores consecutivos no son siempre las mismas.



IEEE-754, Error de precisión por redondeo

- El resultado exacto de una operación v , está entre 2 valores representables v_0 y v_1 .



- Si redondeamos a v_0 , el error de precisión es $\epsilon = |v - v_0|$
- Por ejemplo, si queremos representar el racional $1/10$ en simple precisión:

$$v = 1,100110011001100110011001100110011001100110... \times 2^{-4}$$

23 bits de fracción

- Si lo redondeamos a v_0 (eliminando bits)

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

IEEE-754, Error de precisión por redondeo

- Por ejemplo, si queremos representar el racional $1/10$ en simple precisión:

$$v = 1,100110011001100110011001100110011001100110... \times 2^{-4}$$

← 23 bits de fracción →

- Si lo redondeamos a v_0 (eliminando bits)

$$v_0 = 1,1001100110011001100110011001100110011001100 \times 2^{-4}$$

- Error de precisión ($\epsilon = |v - v_0|$)

$$\epsilon = 0,00000000000000000000000001100110... \times 2^{-4}$$

IEEE-754, Error de precisión por redondeo

- Margen del error absoluto en el intervalo (v_0, v_1) :

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

- Margen del error absoluto en simple precisión

- $v_0 = m \times 2^E$
 - $v_1 = (m+2^{-23}) \times 2^E$
 - $\epsilon_{\max} = |v_1 - v_0| = (m+2^{-23}) \times 2^E - m \times 2^E = 2^{E-23}$

- A veces, el error absoluto no nos dice nada:

- **$\epsilon = 1 \text{ metro}$**
 - ✓ Es aceptable si estamos calculando la distancia entre el sol y la tierra.
 - ✓ Inaceptable si estamos calculando la longitud de un puente.

IEEE-754, Error de precisión por redondeo

- ❑ Error relativo: $\eta = \varepsilon / |v|$
- ❑ Margen del error relativo en simple precisión

- $\eta_{max} < \frac{\varepsilon_{max}}{|v_0|} = \frac{2^{E-23}}{m \cdot 2^E} = \frac{2^{-23}}{m}$

- ❑ Y como $m \geq 1.0$, nos queda que
 - $\eta_{max} < 2^{-23} = 1 \text{ ULP } (\text{Unit in the Last Place})$

IEEE-754, Error de precisión y Underflow

- ❑ Tenemos un caso especial cuando $|v| < 2^{E_{\min}}$
 - $|v|$ está en el intervalo $(v_0, v_1) = (0, 2^{E_{\min}})$
 - Si redondeamos $v_0 = 0$,
 - ✓ el error absoluto es $\mathcal{E} = |v-0| = v$
 - ✓ el error relativo es $\eta = \mathcal{E} / |v| = 1$
- ❑ ¡ El error es tan grande como el resultado !
 - No es fiable
 - El estándar determina que se ha producido un error de **underflow**
 - El tratamiento es configurable: redondear a cero, excepción, ...

IEEE-754, Ejemplo de comportamiento

- Tenemos el siguiente código escrito en C:

```
float x, y;  
x = 0.1;  
y = 0.0;  
for (i=0; i<N; i++)  
    y = y + x;  
  
printf ("%f * %d es: %f \n", x, N, y);
```

- Lo hemos probado con diversos valores de N

```
0.100000 * 10 es: 1.000000  
0.100000 * 100 es: 10.000002  
0.100000 * 1000 es: 99.999046  
0.100000 * 10000 es: 999.902893  
0.100000 * 100000 es: 9998.556641
```

IEEE-754, Codificaciones especiales

- ❑ Se reservan 2 exponentes para casos especiales
 - $E = 000\dots0$
 - $E = 111\dots1$
- ❑ Esto modifica ligeramente los exponentes máximos y mínimos que podemos tener:
 - $E_{\min} = 000\dots01$
 - $E_{\max} = 111\dots10$
- ❑ El rango de exponentes válidos es:
 - $-126 \leq E \leq 127$ para simple precisión (8 bits, exceso a 127)
 - $-1022 \leq E \leq 1023$ para doble precisión (11 bits, exceso a 1023)
- ❑ Casos especiales
 - Zero
 - Infinito
 - Not a Number
 - Denormales

IEEE-754, Codificaciones especiales

CERO

- ❑ No hay ninguna combinación válida de fracción y mantisa que de cero.
- ❑ Usaremos una codificación especial para el cero
 - E = 000...0
 - F = 000000...0
- ❑ Teniendo en cuenta el bit de signo, tenemos dos codificaciones para el cero: +0 y -0
- ❑ En simple precisión el cero es:

s00000000000000000000000000000000

IEEE-754, Codificaciones especiales

INFINITO (“Inf”)

- ❑ Sigue algunas reglas básicas de operación. Algunos ejemplos:

- $\frac{1}{0} = +\infty$, $\frac{1}{\infty} = 0$, $x + \infty = \infty$, etc.

- ❑ Idea muy útil que permite evitar el overflow en algunas expresiones.

- Por ejemplo, $y = \frac{1}{1+\frac{100}{x}}$, $\frac{100}{x}$ generaría un overflow si $x \rightarrow 0$
- Si usamos el infinito, $\frac{100}{\infty} = \infty$ y resulta que $y = 0$.

- ❑ Usaremos una codificación especial para el infinito (Inf y -Inf)

- E = 1111...1
- F = 000000...0

- ❑ En simple precisión el infinito es:

s111111100000000000000000000000000000000



IEEE-754, Codificaciones especiales

NOT A NUMBER (“NaN”)

- ❑ Representa un **resultado inválido**, en algunas operaciones:
 - $\sqrt{-1} = NaN$, $\infty - \infty = NaN$, $\frac{\infty}{\infty} = NaN$, $\log(-1) = NaN$, etc.
 - En general, cualquier operación con un NaN, da como resultado NaN
- ❑ El programa puede comprobar si el resultado es válido o inválido.
 - En una cadena de cálculos podemos diferir la comprobación hasta el final.
- ❑ Codificación de NaN
 - E = 1111...1
 - F ≠ 000000...0
- ❑ En simple precisión el NaN es:

s11111111XXXXXX₀XXXXXXXXXXXXXX

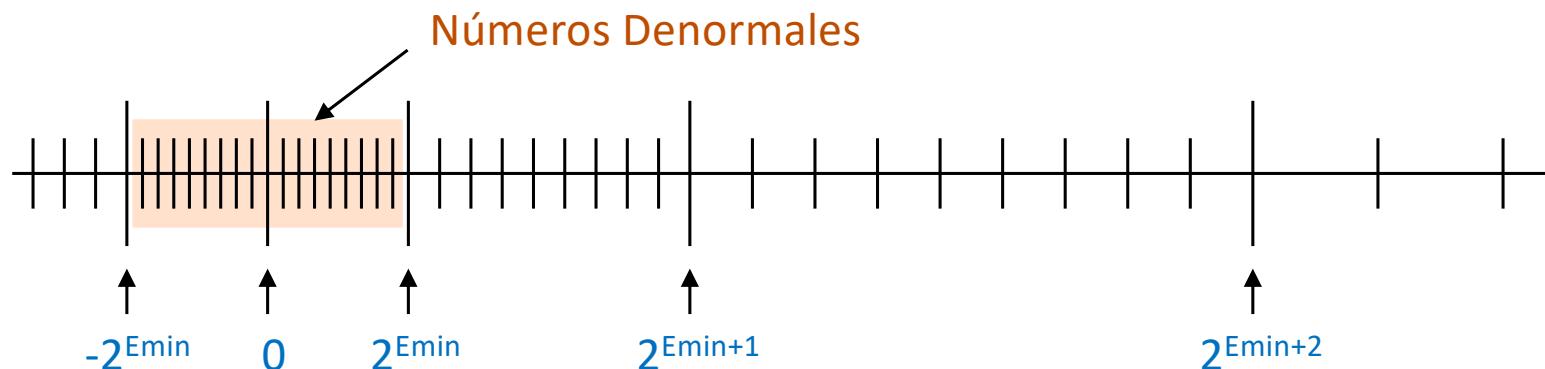
algún X ≠ 0

IEEE-754, Codificaciones especiales

DENORMALES

- Un resultado muy pequeño, del tipo $|v| < 2^{E_{\min}}$ no es fiable, ya que si lo redondeamos a 0, el error de precisión es enorme:

- $\epsilon = |v - 0| = |v|$ $\eta = \frac{|v-0|}{|v|} = 1$



- En estos casos podemos mejorar la precisión admitiendo que existan números **NO NORMALIZADOS o DENORMALES** en el rango $(-2^{E_{\min}}, 0, 2^{E_{\min}})$:

$$v = (-1)^s \times 0,FFFF...F \times 2^{E_{\min}}$$

IEEE-754, Codificaciones especiales

DENORMALES

- Admitimos números **NO NORMALIZADOS** o **DENORMALES** en el rango $(-2^{E_{\min}}, 0, 2^{E_{\min}})$:

$$v = (-1)^s \times 0,FFFF...F \times 2^{E_{\min}}$$

- En simple precisión, el margen superior de error absoluto es:

- $\varepsilon = |v_1 - v_0| = 2^{E_{\min} - 23}$

- Codificación de Denormales

- E = 000...0
 - F \neq 000000...0

- En simple precisión es:

¡Atención!

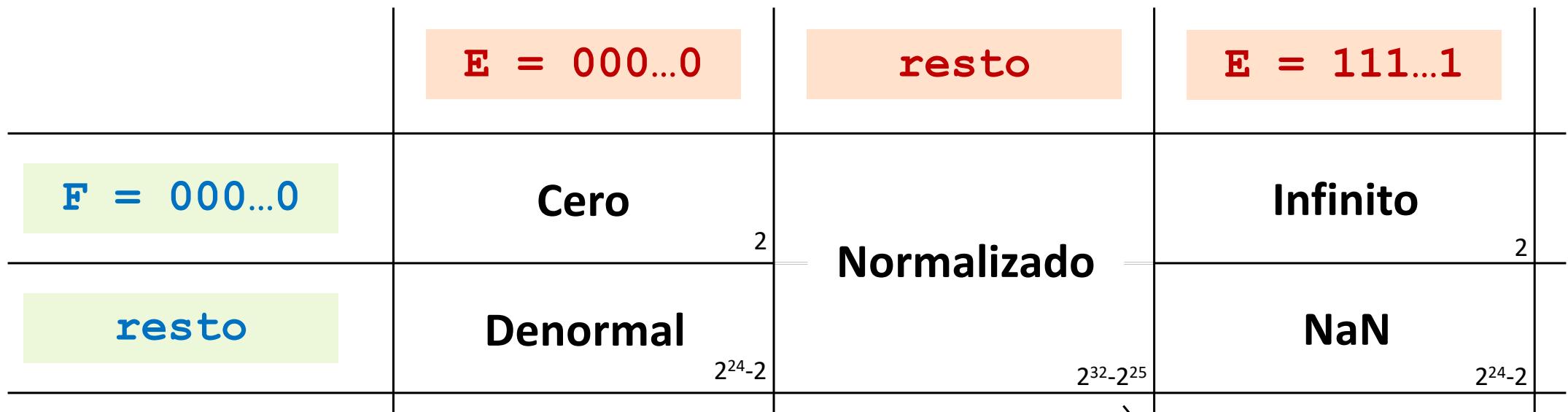
Si operamos con denormales:

- El bit oculto implícito es 0.
- El exponente implícito es E_{\min} .

S00000000XXXXXXXXXXXXXXXXXXXX

algún X \neq 0

IEEE-754, Resumen de formatos



#valores de cada tipo
en precisión simple (32b)

Resumiendo

IEEE-754

□ **float** (precisión simple, 32b, 4B)

- 1 bit de signo, 8 bits de exponente (exceso 127) y 23 bits de mantisa
- Rango:
 - ✓ $(\pm) 1,18 \cdot 10^{-38} \leq x \leq 3,4 \cdot 10^{38}$
 - ✓ aproximadamente 7 dígitos decimales de precisión

□ **double** (precisión doble, 64b, 8B)

- 1 bit de signo, 11 bits de exponente (exceso 1023) y 52 bits de mantisa
- Rango:
 - ✓ $(\pm) 2,23 \cdot 10^{-308} \leq x \leq 1,8 \cdot 10^{308}$
 - ✓ aproximadamente 15 dígitos decimales de precisión

Un experimento

0,1	0,00011	
$0,1 \times 2$	$= 0,2$	0
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
...		

0,2	0,0011	
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
...		

0,3	0,01001	
$0,3 \times 2$	$= 0,6$	0
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
...		

0,4	0,0110	
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
...		

0,5	0,1	
$0,5 \times 2$	$= 1,0$	1
$0,0 \times 2$	$= 0,0$	0

0,1 0,00011 ...

0,2 0,0011

0,3 0,01001 ...

0,4 0,0110

0,5 0,1

0,6 0,1001

0,7 0,10110 ...

0,8 0,1100

0,9 0,11100 ...

0,6	0,1001	
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
...		

0,7	0,10110	
$0,7 \times 2$	$= 1,4$	1
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
...		

0,8	0,1100	
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
...		

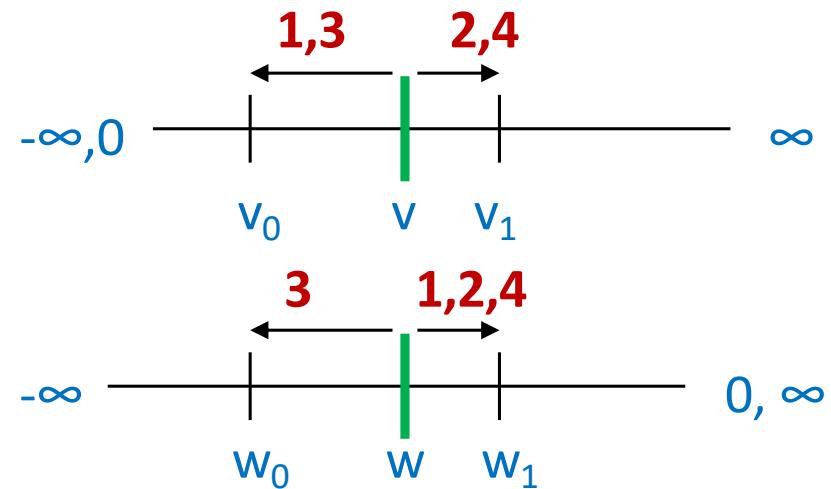
0,9	0,11100	
$0,9 \times 2$	$= 1,8$	1
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
$0,2 \times 2$	$= 0,4$	0
$0,4 \times 2$	$= 0,8$	0
$0,8 \times 2$	$= 1,6$	1
$0,6 \times 2$	$= 1,2$	1
...		



IEEE-754, Modos de Redondeo

- ❑ Cuando tenemos un número v , no representable de forma exacta, hemos de redondearlo a un número cercano:
 - $v_0 < v < v_1$
 - v_0 y v_1 son números consecutivos

- ❑ En el estándar IEEE-754 tenemos 4 modos de redondeo que podemos escoger:
 1. Hacia cero
 2. Hacia $+\infty$
 3. Hacia $-\infty$
 4. Al más próximo
 - ✓ Es el modo por defecto



IEEE-754, Modos de Redondeo

□ Hacia Cero (Truncado):

- $0 < v_0 < v < v_1$
- $v \rightarrow v_0$
- Es el más fácil de implementar, sólo hay que eliminar los bits que sobran (truncar)
- El margen de error en simple precisión es:
 - ✓ $\mathcal{E}_{\max} < |v_1 - v_0| = 2^{-23}$
 - ✓ $\mathcal{E}_{\max} < 2^{-23}$
 - ✓ $\eta_{\max} < 2^{-23} = 1 \text{ ULP}$

IEEE-754, Modos de Redondeo

- ❑ Hacia +Infinito:
 - $v_0 < v < v_1$
 - $v \rightarrow \max(v_0, v_1)$
- ❑ Hacia -Infinito:
 - $v_0 < v < v_1$
 - $v \rightarrow \min(v_0, v_1)$
- ❑ Este tipo de redondeo se utiliza cuando trabajamos con aritmética de intervalos:
 - Altura media = $1,75 \pm 5$ cm

IEEE-754, Modos de Redondeo

□ Hacia el más próximo:

- Es el método usado por defecto, porque da el menor error posible
- Hacia el par si v está justo en medio
- $v_0 < v < v_1$, si v es equidistante es cuando se produce el máximo error
 - ✓ $\mathcal{E}_{\max} < |v_1 - v_0|/2$
 - ✓ $\mathcal{E}_{\max} < 2^{E-24}$
 - ✓ $\eta_{\max} < 2^{-24} = 0.5 \text{ ULP}$
- ¿Cómo se hace?

IEEE-754, Modos de Redondeo

- Regla práctica para redondear al más próximo:
 - Calculamos el resultado con algunos bits extra de precisión
 - Examinamos el **primer bit extra de la mantisa** y tenemos 3 casos:

a) El bit es 0, redondeamos al anterior (v_0):

$$\begin{aligned} v &= 1.\text{xxxxx...0011} \quad 0000101 \\ v_0 &= 1.\text{xxxxx...0011} \end{aligned}$$

b) El bit es 1, y el resto no son todo ceros, redondeamos al siguiente (v_1):

$$\begin{aligned} v &= 1.\text{xxxxx...0011} \quad 1010110 \\ &\quad + 0.00000...0001 \\ v_1 &= 1.\text{xxxxx...0100} \end{aligned}$$

c) El bit es 1, y el resto son todos cero

$$v = 1.\text{xxxxx...0011} \quad 1000000$$

v es equidistante de v_0 y v_1 , redondeamos al que sea par

$$\begin{aligned} v_0 &= 1.\text{xxxxx...0011} \\ v_1 &= 1.\text{xxxxx...0100} \leftarrow \text{"par"} \end{aligned}$$

Conversión de base 10 a base 2

Queremos representar $v = -1029,68$

1. Convertir la parte entera, por divisiones sucesivas

$$1029 = 10000000101 \text{ (necesitamos 11 bits)}$$

2. Convertir la fracción, por productos sucesivos:

$$0,68 \times 2 = 1,36 \rightarrow 1$$

$$0,36 \times 2 = 0,72 \rightarrow 0$$

$$0,72 \times 2 = 1,44 \rightarrow 1$$

...

- ¿Cuántos bits calculamos? → 13 bits para completar los 24 bits de la mantisa

$$0,68 = 0,1010111000010 \text{ (13 bits)}$$

y algunos bits extra para decidir cómo redondeamos

$$0,68 = 0,101011100001010001 \text{ (18 bits)}$$

Conversión de base 10 a base 2

3. Juntar la parte entera y la fracción ($11+18 = 29$ bits)

$$1029,68 = 10000000101,101011100001010001$$

4. Normalizar, mover la coma hacia la izquierda para que la mantisa sea 1,xxx

$$1029,68 = 1,0000000101101011100001010001 \times 2^{10}$$

5. Redondeamos al más próximo usando los bits extra,

→ redondeamos al siguiente

$$1029,68 = 1,00000001011010111000011 \times 2^{10}$$

6. Codificar el exponente en exceso a 127

$$E = 10 + 127 = 137 = 10001001$$

7. Juntar Signo, Exponente y Fracción

1 10001001 00000001011010111000011

8. Expresar el número en hexadecimal

-1029,68 = 0xC480B5C3

Conversión de base 10 a base 2

- Podemos calcular el error de precisión ($\varepsilon = |v - v_0|$)

Restando el valor redondeado y el exacto

$$\begin{aligned}\varepsilon &= 1,0000001011010111000011 \times 2^{10} \\ &\quad - 1,000000101101011100001010001 \times 2^{10} \\ &= 0,000000000000000000000001111 \times 2^{10}\end{aligned}$$

Normalizamos, movemos la coma 25 posiciones a la derecha

$$\begin{aligned}\varepsilon &= 00000000000000000000000000001,111 \times 2^{10-25} \\ &= 1,111 \times 2^{-15}\end{aligned}$$

Lo pasamos a decimal (no se pedirá sin calculadora),

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

Conversión de binario (coma flotante) a base 10

Queremos saber que valor tiene $v = 0x45814140$

1. Escribirlo en binario

$$v = 0100\ 0101\ 1000\ 0001\ 0100\ 0001\ 0100\ 0000$$

2. Identificamos los 3 campos: signo, exponente, fracción

$$v = 0\ 10001011\ 0000010100000101000000$$

3. Exponente. Lo pasamos a decimal y le restamos el exceso (127)

$$10001011 = 139$$

$$E = 139 - 127 = 12$$

4. Componer el número: signo, bit oculto, fracción y exponente

$$v = +1.\underline{0000001010000010100000} \times 2^{12}$$

5. Punto Fijo. Mover la coma 12 posiciones a la derecha y eliminar ceros

$$v = +1\underline{000000101000.0010100000}$$

Conversión de binario (coma flotante) a base 10

6. Convertir la parte entera a base 10

$$1000000101000 = 1 \cdot 2^{12} + 1 \cdot 2^5 + 1 \cdot 2^3 = 4136$$

7. Convertir la fracción a base 10 (1º desplazaremos la coma)

$$0.00101 = 101 \times 2^{-5} = 5/32 = 0,15625$$

8. Finalmente, juntar la parte entera y la fracción:

$$v = 4136,15625$$

Operaciones en Coma Flotante: Suma y Resta

- Primero recordaremos la suma que ya conocemos (base 10):

- Formato: mantisa normalizada 4 dígitos: $z.zzz \times 10^{zz}$
 - Sumar: $9,999 \times 10^1 + 1,680 \times 10^{-1}$
 - ¿Cómo lo hacemos?

$$\begin{array}{r} 9,999 \times 10^1 \\ + 1,680 \times 10^{-1} \\ \hline = 11,679 \times 10^? \end{array} \quad \leftarrow \textcolor{red}{\text{¡Así, NO se hace!}}$$

- Igualar exponentes (al mayor, =1), alinear mantisas y sumar:

$$\begin{array}{r} 9,99900 \times 10^1 \\ + 0,01680 \times 10^1 \\ \hline = 10,01580 \times 10^1 \end{array}$$

- Normalizar:

$$= 1,001580 \times 10^2$$

- Redondear a 4 dígitos:

$$= 1,002 \times 10^2$$

Operaciones en Coma Flotante: Suma (y Resta)

1. Igualar exponentes, al mayor de los 2
 - o Alinear las mantisas, desplazando a la izquierda la coma de la mantisa con menor exponente
2. Sumar las magnitudes (valor absoluto)
 - o Si signos iguales \Rightarrow sumar magnitudes
 - o Si signos diferentes \Rightarrow restar la magnitud mayor menos la menor, y asignar el signo de la mayor al resultado
3. Normalizar el resultado
 - o Moviendo la coma para obtener un dígito $\neq 0$ en la parte entera
4. Redondear la mantisa
 - o Al valor representable más próximo
 - o Puede provocar que se deba normalizar y redondear de nuevo
5. Codificar el resultado
 - o Signo, exponente (en exceso) y mantisa (sin el bit oculto)

Ejemplo de suma en Coma Flotante: $z = x + y$

Suponemos $x = 0x3F40000D$, e $y = 0xC0800004$

1. Los escribimos en binario

$$\begin{aligned}x &= 0011\ 1111\ 0100\ 0000\ 0000\ 0000\ 0000\ 1101 \\y &= 1100\ 0000\ 1000\ 0000\ 0000\ 0000\ 0000\ 0100\end{aligned}$$

2. Identificamos los 3 campos: signo, exponente, fracción

$$\begin{aligned}x &= 0\ 01111110\ 1000000000000000000000001101 \\y &= 1\ 10000001\ 000000000000000000000000100\end{aligned}$$

3. Obtenemos los exponentes en base 10 (restando el exceso 127)

$$01111110 = 126 \Rightarrow E = 126 - 127 = -1$$

$$10000001 = 129 \Rightarrow E = 129 - 127 = 2$$

4. Componer x e y: signo, bit oculto, fracción y exponente

$$x = +1.\underbrace{100000000000000000000000}_{\text{fracción}}1101 \times 2^{-1}$$

$$y = -1.\underbrace{000000000000000000000000}_{\text{fracción}}0100 \times 2^2$$

Ejemplo de suma en Coma Flotante: $z = x + y$

5. Igualamos exponentes ($2^{-1} \Rightarrow 2^2$), desplazando la coma 3 bits a la izquierda

$$\begin{aligned}x &= +0,001\textcolor{blue}{10000000000000000000000001101} \times 2^2 \quad (\text{en rojo, bits extra}) \\y &= -1,\textcolor{blue}{0000000000000000000000000000000100} \times 2^2\end{aligned}$$

6. Como tenemos signos diferentes \Rightarrow restamos la magnitud mayor (y) menos la menor (x)

$$\begin{aligned}|y| &= 1,\textcolor{blue}{0000000000000000000000000000000100} \times 2^2 \\-|x| &= 0,\textcolor{blue}{00110000000000000000000000000001101} \times 2^2 \\|z| &= 0,\textcolor{blue}{110100000000000000000000000000010011} \times 2^2\end{aligned}$$

7. Asignar el signo del que tiene mayor valor absoluto (y, negativo)

$$z = -0,\textcolor{blue}{110100000000000000000000000000010011} \times 2^2$$

8. Normalizar mantisa (desplazando bits si es necesario)

$$z = -1,\textcolor{blue}{101000000000000000000000000000010011} \times 2^1$$

9. Redondear mantisa al más cercano

$$z = -1,\textcolor{blue}{1010000000000000000000000000000101} \times 2^1$$

Ejemplo de suma en Coma Flotante: $z = x + y$

10. Codificar el exponente (en exceso)

$$E = 1+127 = 128 = 10000000$$

11. Juntar signo exponente y fracción (sin bit oculto)

S: 1

E: 10000000

F: 1010000000000000000000101

$$\begin{aligned} z &= 1 \text{ } 10000000 \text{ } 1010000000000000000000101 = \\ &= 1100 \text{ } 0000 \text{ } 0101 \text{ } 0000 \text{ } 0000 \text{ } 0000 \text{ } 0000 \text{ } 0101 = \\ &= 0xC0500005 \end{aligned}$$

Bits de Guarda

- Cuando igualamos exponentes, ¿cuántos bits hay que desplazar la mantisa en el peor caso?
 - Por ejemplo, si restamos $x = 1,0 \times 2^{127}$ menos $y = 1,0 \times 2^{-126}$, habría que mover la coma de y , 253 posiciones a la izquierda

$$x = 1,000\dots0 \times 2^{127}$$

23 bits

$$y = 0,000000\dots001 \quad 000\dots0 \times 2^{127}$$

253 posiciones 23 bits

- Para no perder precisión, necesitaríamos un sumador con casi 300 bits \Rightarrow **NO ES LA SOLUCIÓN**
- Usaremos los bits de guarda G, R y S

Bits de Guarda

- Usaremos sólo 3 bits de guarda:

- **Guard (G)**: bit 24 de la mantisa
 - **Round(R)**: bit 25 de la mantisa
 - **Sticky (S)**: OR lógica de todos los bits a la derecha del bit 25

- Ejemplo:

$$x = 1,00000000000000000000000000 \times 2^5$$

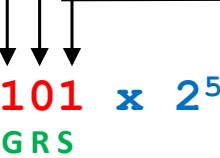
$$y = 1,00110011001100110011011 \times 2^{-3}$$

- Igualamos exponentes

$$y = 0,0000001001100110011001\textcolor{red}{10011011} \times 2^5$$

- Operaremos con 3 bits de guarda: G, R, S

$$y = 0,0000001001100110011001\textcolor{red}{101} \times 2^5$$



- Usando estos 3 bits de guarda obtendremos los mismos resultados que si usáramos infinitos bits

Multiplicación y División en Coma Flotante

- Sean x, y tal que:
 - $x = m_x \times 2^{ex}$
 - $y = m_y \times 2^{ey}$
- El producto y el cociente son:
 - $x \times y = (m_x \times 2^{ex}) \times (m_y \times 2^{ey}) = (m_x \times m_y) \times 2^{(ex+ey)}$
 - $x / y = (m_x \times 2^{ex}) / (m_y \times 2^{ey}) = (m_x / m_y) \times 2^{(ex-ey)}$
- Algoritmo:
 - Multiplicar (o dividir) las mantisas (igual que con números naturales)
 - Sumar (o restar) exponentes
 - Ajustar el signo: positivo si son iguales, negativo en caso contrario
 - Normalizar y redondear la mantisa

Ejemplo de Multiplicación en base 10

- ❑ Mantisa normalizada con 4 dígitos: Z.ZZZ × 10^{zz}.
- ❑ Multiplicar $(-1,110 \times 10^{10}) \times (9,200 \times 10^{-5})$
 - Producto de mantisas: $1,11 \times 9,2 = 10,212$
 - Suma de exponentes: $10 + (-5) = 5$
 - Normalizar: $10,212 \times 10^5 \Rightarrow 1,0212 \times 10^6$
 - Redondear: $1,021\textcolor{red}{2} \times 10^6 \Rightarrow 1,021 \times 10^6$
 - Ajustar signo (negativo): $-1,021 \times 10^6$
- ❑ Dividir $(9,030 \times 10^{10}) / (2,100 \times 10^{-5})$
 - Cociente de mantisas: $9,03 / 2,1 = 4,3$
 - Resta de exponentes: $10 - (-5) = 15$
 - Normalizar: $4,300 \times 10^{15}$
 - Redondear: $4,300 \times 10^{15}$
 - Ajustar signo: $4,300 \times 10^{15}$

Ejemplo de producto en Coma Flotante: $z = x \times y$

Suponemos $x = 0x3F600000$, e $y = 0xBED00002$

1. Los escribimos en binario

$$\begin{aligned}x &= 0011\ 1111\ 0110\ 0000\ 0000\ 0000\ 0000\ 0000 \\y &= 1011\ 1110\ 1101\ 0000\ 0000\ 0000\ 0000\ 0010\end{aligned}$$

2. Identificamos los 3 campos: signo, exponente, fracción

$$\begin{aligned}x &= 0\ 01111110\ 11000000000000000000000000000000 \\y &= 1\ 01111101\ 10100000000000000000000000000000\end{aligned}$$

3. Obtenemos los exponentes en base 10 (restando el exceso 127)

$$01111110 = 126 \Rightarrow E = 126 - 127 = -1$$

$$01111101 = 125 \Rightarrow E = 125 - 127 = -2$$

4. Componer x e y: signo, bit oculto, fracción y exponente

$$x = +1.1100000000000000000000000000000 \times 2^{-1}$$

$$y = -1.1010000000000000000000000000000 \times 2^{-2}$$

Ejemplo de producto en Coma Flotante: $z = x \times y$

5. Multiplicamos mantisas, sólo mostramos los distintos de cero

	111000000000000000000000
×	110100000000000000000010
	111000000000000000000000
	111000000000000000000000
	111000000000000000000000
	111000000000000000000000
+	111000000000000000000000
	101101100000000000000000

6. Suma de exponentes: $-1 + (-2) = -3$

- ## 7. Normalizar

- ## 8. Redondear (↑)

Ejemplo de producto en Coma Flotante: $z = x \times y$

9. Codificamos el exponente (exceso a 127)

$$-2 + 127 = 125 = 01111101$$

10. Juntar signo (-), exponente y mantisa (sin bit oculto)

$$\begin{aligned} z &= 1 \text{ } 01111101 \text{ } 011011000000000000000010 \\ &= 1011 \text{ } 1110 \text{ } 1011 \text{ } 0110 \text{ } 0000 \text{ } 0000 \text{ } 0000 \text{ } 0010 \\ &= 0xBEB60002 \end{aligned}$$

Coma Flotante en MIPS

- ❑ Históricamente, la Unidad de Coma Flotante (FPU) era un chip opcional, separado de la CPU. Le llamábamos coprocesador.
- ❑ El ISA de MIPS está organizado así, aunque ahora la Unidad de Coma Flotante ya se construye integrada en la CPU.
- ❑ En MIPS, tenemos el CP1 (coprocesador 1) que es la Unidad de Coma Flotante
 - Banco de registros propio. 32 registros de 32 bits: \$f0 ... \$f31
 - O bien 16 registros dobles: \$f0-\$f1, \$f2-\$f3 ... \$f30-\$f31 (se identifican con el par)
 - Todas las operaciones aritméticas en coma flotante se realizan con registros \$fxx.
 - Puede operar con floats (32 bits, 4B) o doubles (64 bits, 8B).
 - Dispone de un registro de control adicional para configurar el redondeo, gestionar excepciones, comparaciones, ...
 - Dispone de instrucciones especiales para mover datos entre los registros de la CPU y los registros del coprocesador

Intrucciones MIPS de Coma Flotante

□ Acceso a Memoria

Simple Precisión (float)	Doble Precisión (double)	Comportamiento
<code>lwcl ft, offset(rs)</code>	<code>ldcl ft, offset(rs)</code>	load de memoria
<code>swcl ft, offset(rs)</code>	<code>sdc1 ft, offset(rs)</code>	store en memoria

```
lwcl $f3, 40($t2)      # $f3 = M[$t2+40], leemos 4 bytes  
swcl $f4, 10($t3)       # M[$t3+10] = $f4, escribimos 4 bytes  
  
ldcl $f2, 40($t2)       # $f3:$f2 = M[$t2+40], leemos 8 bytes  
sdc1 $f8, 10($t3)       # M[$t3+10] = $f9:$f8, escribimos 8 bytes
```

Intrucciones MIPS de Coma Flotante

□ Aritméticas

Simple Precisión (float)	Doble Precisión (double)	Comportamiento (float)
<code>add.s fd,fs,ft</code>	<code>add.d fd,fs,ft</code>	$fd = fs + ft$
<code>sub.s fd,fs,ft</code>	<code>sub.d fd,fs,ft</code>	$fd = fs - ft$
<code>mul.s fd,fs,ft</code>	<code>mul.d fd,fs,ft</code>	$fd = fs * ft$
<code>div.s fd,fs,ft</code>	<code>div.d fd,fs,ft</code>	$fd = fs / ft$

□ Movimiento de registros

Instrucción	Comportamiento
<code>mfcl rt, fs</code>	$rt = fs$
<code>mtcl rt, fs</code>	$fs = rt$
<code>mov.s fd, fs</code>	$fd = fs$

Instrucciones MIPS de Coma Flotante

□ Comparación

Simple Precisión (float)	Doble Precisión (double)	Comportamiento (float)
<code>c.eq.s fs,ft</code>	<code>c.eq.d fs,ft</code>	$cc = (fs == ft)$
<code>c.lt.s fs,ft</code>	<code>c.lt.d fs,ft</code>	$cc = (fs < ft)$
<code>c.le.s fs,ft</code>	<code>c.le.d fs,ft</code>	$cc = (fs \leq ft)$

- El resultado de la comparación se escribe en un **bit de condición (cc)** de un registro interno

□ Instrucciones de salto

Instrucción	Comportamiento
<code>bc1t etiq</code>	salta si $cc = \text{TRUE} (1)$
<code>bc1f etiq</code>	salta si $cc = \text{FALSE} (0)$

- Es un mecanismo común

Declaraciones MIPS de Coma Flotante

- Declaración de variables GLOBALES en coma flotante

Tipo C	MIPS	Tamaño
float	.float	4 bytes (1 word)
double	.double	8 bytes (2 words)

- Las directivas float y double alinean a 4 y 8 bytes respectivamente
- Ejemplo

```
float v[2] = {3.1416, -3.5E2};  
double x = 3E350, y;
```

```
.data  
v: .float 3.1416, -3.5E2  
x: .double 3E350  
y: .double 0.0
```

Declaraciones globales en C

Declaraciones en MIPS

Subrutinas con Coma Flotante

- Paso de parámetros y resultados
 - Parámetros en **\$f12** y **\$f14**
 - Resultado en **\$f0**
 - Registros Seguros: del **\$f20** al **\$f31**
- Ejemplo de subrutina

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

NOTA: La mezcla de parámetros en coma flotante con enteros, sigue en MIPS una reglas muy complejas que no estudiaremos en EC. Sólo estudiaremos un caso: cuando tengamos exclusivamente 1 o 2 parámetros de tipo **float**.

Subrutinas con Coma Flotante, ejemplo de traducción

```
.data  
uno: .float 1.0
```

```
.text
```

```
...
```

```
func: la $t0,uno  
      lwc1 $f16,0($t0)          # f16 = 1.0  
      c.lt.s $f12,$f16          # ¿x<1.0?  
      bclt else                # si false  
      mul.s $f0,$f12,$f12       # x*x  
      b end  
else: add.s $f16,$f16,$f16    # f16 = 2.0  
      sub.s $f0,$f16,$f12        # 2.0-x  
end:  jr $ra
```

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

Guardamos la constante 1.0 en memoria. No hay instrucciones que operen con inmediatos

Subrutinas con Coma Flotante, ejemplo de traducción

```
.data
uno: .float 1.0
.text
...
func: la $t0,uno
      lwc1 $f16,0($t0)          # f16 = 1.0
      c.lt.s $f12,$f16          # ¿x<1.0?
      bclif else                # si false
      mul.s $f0,$f12,$f12       # x*x
      b end
else: add.s $f16,$f16,$f16    # f16 = 2.0
      sub.s $f0,$f16,$f12       # 2.0-x
end: jr $ra
```

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

Cargamos la constante
1.0 en \$f16

Subrutinas con Coma Flotante, ejemplo de traducción

```
.data
uno: .float 1.0
.text
...
func: la $t0,uno
      lwc1 $f16,0($t0)          # f16 = 1.0
      c.lt.s $f12,$f16          # ¿x<1.0?
      bclt else                # si false
      mul.s $f0,$f12,$f12       # x*x
      b end
else: add.s $f16,$f16,$f16    # f16 = 2.0
      sub.s $f0,$f16,$f12       # 2.0-x
end:  jr $ra
```

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

si ($x < 1.0$) es falso,
saltamos al else

Subrutinas con Coma Flotante, ejemplo de traducción

```
.data
uno: .float 1.0
.text
...
func: la $t0,uno
      lwc1 $f16,0($t0)          # f16 = 1.0
      c.lt.s $f12,$f16          # ¿x<1.0?
      bclt else                 # si false
      mul.s $f0,$f12,$f12       # x*x
      b end
else: add.s $f16,$f16,$f16    # f16 = 2.0
      sub.s $f0,$f16,$f12       # 2.0-x
end:  jr $ra
```

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

El parámetro x está en \$f12, dejamos el resultado en \$f0, acabamos

Subrutinas con Coma Flotante, ejemplo de traducción

```
.data
uno: .float 1.0
.text
...
func: la $t0,uno
      lwc1 $f16,0($t0)          # f16 = 1.0
      c.lt.s $f12,$f16          # ¿x<1.0?
      bclt else
      mul.s $f0,$f12,$f12      # x*x
      b end
else: add.s $f16,$f16,$f16    # f16 = 2.0
      sub.s $f0,$f16,$f12      # 2.0-x
end:  jr $ra
```

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

Calculamos la constante 2.0, dejamos el resultado en \$f0 y acabamos

Asociatividad de la Suma en Coma Flotante

- La suma de números en coma flotante **NO CUMPLE** la propiedad **ASOCIATIVA**:

$$x + (y + z) \neq (x + y) + z$$

- Un ejemplo muy simple: $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$
 - $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = -1.5 \times 10^{38} + 1.5 \times 10^{38} = 0.0$
 - $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = 0.0 + 1.0 = 1.0$
- Es un problema provocado por la coma flotante y no tiene solución.
- ¡Mucha atención cuando tengamos aplicaciones paralelas!

Asociatividad de la Suma en Coma Flotante

- La suma de números en coma flotante **NO CUMPLE** la propiedad **ASOCIATIVA**:

$$x + (y + z) \neq (x + y) + z$$

- Las diferencias no tienen porque ser muy grandes, pero hay cosas a evitar

```
if (suma == 1.0)
    ...
```

- Es una construcción “peligrosa”. Si hay que comprobar un resultado, hay que calcular el error máximo aceptable y ver si el resultado está dentro de ese intervalo: $1-0\pm\text{epsilon}$

```
if (abs(suma-1.0) <= epsilon)
    ...
```

Ejercicio: Sumar $z = x + y$

Suponemos $x = 0x3DC00046$, e $y = 0xC0800004$

1. Los escribimos en binario

$$\begin{aligned}x &= 0011\ 1101\ 1100\ 0000\ 0000\ 0000\ 0100\ 0110 \\y &= 1100\ 0000\ 1000\ 0000\ 0000\ 0000\ 0000\ 0100\end{aligned}$$

2. Identificamos los 3 campos: signo, exponente, fracción

$$\begin{aligned}x &= 0\ \textcolor{red}{01111011}\ 10000000000000001000110 \\y &= 1\ \textcolor{red}{10000001}\ 000000000000000000000000100\end{aligned}$$

3. Obtenemos los exponentes en base 10 (restando el exceso 127)

$$\textcolor{red}{01111011} = 123 \Rightarrow E = 123 - 127 = -4$$

$$\textcolor{red}{10000001} = 129 \Rightarrow E = 129 - 127 = 2$$

4. Componer x e y: signo, bit oculto, fracción y exponente

$$x = +1.\textcolor{blue}{10000000000000001000110} \times 2^{-4}$$

$$y = -1.\textcolor{blue}{000000000000000000000000100} \times 2^2$$

Ejemplo de suma en Coma Flotante: $z = x + y$

5. Igualamos exponentes ($2^{-4} \Rightarrow 2^2$), desplazando la coma 6 posiciones a la izquierda

$$\begin{aligned}x &= +0.00000110000000000000001000110 \times 2^2 && \text{(en rojo, bits extra)} \\y &= -1.0000000000000000000000000000000100\end{aligned}$$

OR

6. Determinamos los bits de guarda G, R, S

$$x = +0.00000110000000000000001001 \times 2^2$$

7. Sumar magnitudes. Signos diferentes \Rightarrow Restar del mayor: $|y| - |x|$

$$\begin{array}{r} |y|=1.0000000000000000000000000000000100000 \times 2^2 \\ - |x|=0.00000110000000000000000000000001001 \times 2^2 \\ \hline |z|=0.111101000000000000000000000000010111 \times 2^2 \end{array}$$

8. Normalizar mantisa (desplazando bits si es necesario)

$$|z|=1.111010000000000000000000000000010111 \times 2^1$$

9. Redondear mantisa al más cercano (\uparrow)

$$|z|=1.1110100000000000000000000000000101 \times 2^1$$

+1

$$|z|=1.1110100000000000000000000000000110 \times 2^1$$

Ejemplo de suma en Coma Flotante: $z = x + y$

10. Codificar el exponente (en exceso)

$$E = 1+127 = 128 = 10000000$$

11. Juntar signo exponente y fracción (sin bit oculto)

S: 1

E: 10000000

F: 111101000000000000000101

$$\begin{aligned} z &= 1 \text{ } 10000000 \text{ } 111101000000000000000110 = \\ &= 1100 \text{ } 0000 \text{ } 0111 \text{ } 1010 \text{ } 0000 \text{ } 0000 \text{ } 0000 \text{ } 0110 = \\ &= 0xC07A0006 \end{aligned}$$



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Estructura de Computadores

Tema 5: Aritmética de Enteros y Coma Flotante

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

