Tema 3

La aritmética en el computador

§ Tema 3. Capítulo 3 del libro

La aritmética en el computador

- **Operaciones con enteros**
  - Adición y sustracción
  - Multiplicación y división
  - Tratamiento del desbordamiento (*overflow*)
- **Números reales en punto flotante**
  - Representación y operaciones

Adición de enteros

- **Ejemplo: 7 + 6**

		(0)		*(0)*		*(1)*		*(1)*		*(0)*		*(Carries)*
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- **Hay desbordamiento (*overflow*) si el resultado se sale de rango (no cabe en el registro destino):**
  - Sumando operandos de distinto signo no hay *overflow*
  - Hay desbordamiento si se suman operandos del mismo signo y el bit de signo del resultado es diferente

Sustracción de enteros

- **Se suma al minuendo el opuesto al sustraendo**
  - Ejemplo: $7 - 6 = 7 + (-6)$

```
+7:      0000 0000 … 0000 0111
-6:      1111 1111 … 1111 1010
+1:      0000 0000 … 0000 0001
```

- **Hay desbordamiento si el resultado se sale de rango:**
  - Restando dos operandos del mismo signo no hay desbordamiento
  - Restando un operando positivo de uno negativo:
    - Hay desbordamiento si el signo del resultado es 0
  - Restando un operando negativo de uno positivo:
    - Hay desbordamiento si el signo del resultado es 1
- **RESUMEN:**
  Hay desbordamiento en la sustracción a-b si lo hay en la suma $a + (-b)$

Tratamiento del desbordamiento

- **Algunos lenguajes (por ejemplo, C) ignoran el desbordamiento**
  - En esos casos se usan las instrucciones de MIPS `addu`, `addui`, `subu`
- **Otros lenguajes (por ejemplo, Ada o Fortran) provocan una excepción**
  - Entonces se usan las instrucciones de MIPS `add`, `addi`, `sub`
  - En caso de desbordamiento, se recurre al manejador de excepciones
    - Guarda el PC en el registro EPC (*exception program counter*)
    - Salta a la dirección de la rutina de tratamiento de excepción
    - La instruction `mfc0` (*move from coprocessor reg*) recupera el valor del EPC en el PC, para retornar al programa después de la ejecución de la rutina de tratamiento de excepción

Aritmética para Multimedia

- **El proceso de datos multimedia (imagen y sonido) necesita operar con vectores muy grandes de datos de 8 y 16 bits.**
  - Se puede usar un sumador de 64 bits con la llevada (*carry*) particionada.
    - Podría operarse con vectores de 8×8 bits, 4×16 bits o 2×32 bits
  - Ejemplo de SIMD (*single-instruction, multiple-data*)
- **Operaciones con saturación**
  - En caso de *overflow*, el resultado es el mayor valor representable
  - Ejemplos: recorte del audio o saturación en video

Multiplicación

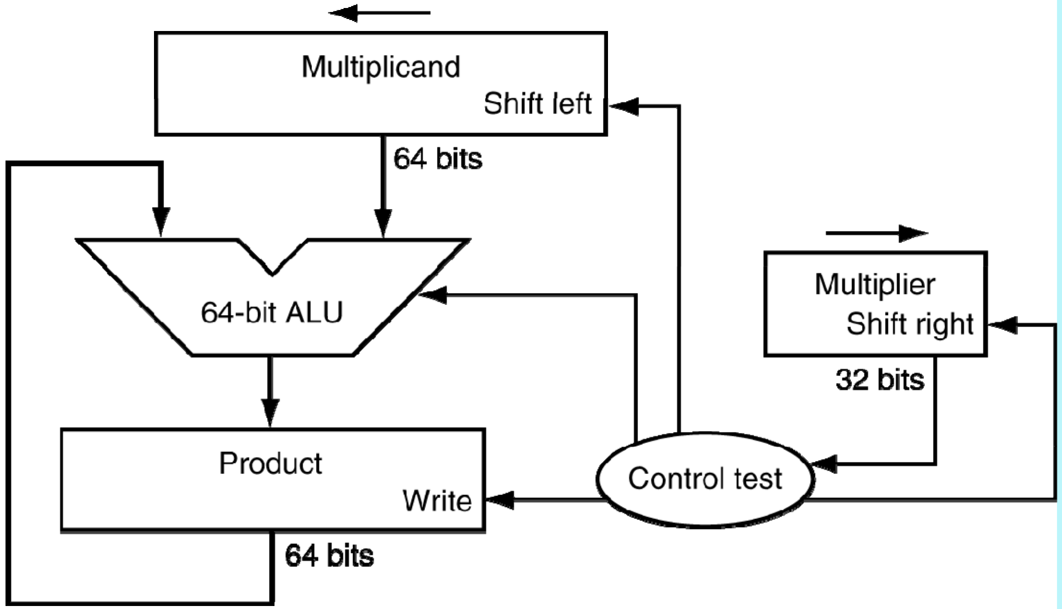
Algoritmo de "lápiz y papel"

Multiplicando

Multiplicador

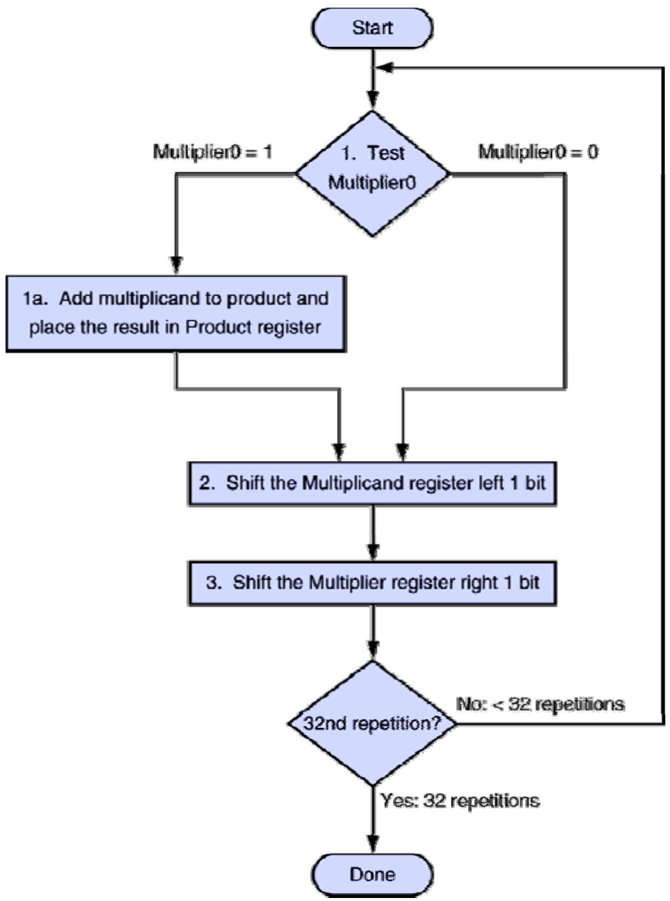
```
      1000
   ×  1001
   -------
      1000
     0000
    0000
   1000
   -------
   1001000
```

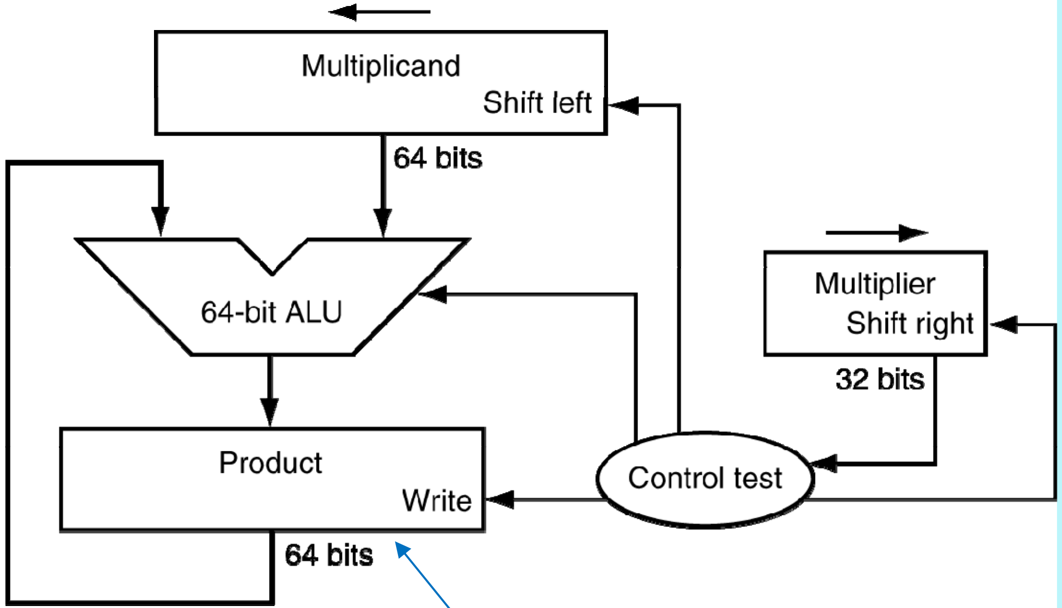
Producto



La longitud del producto es la suma de las longitudes de los operandos

Multiplicación por hardware

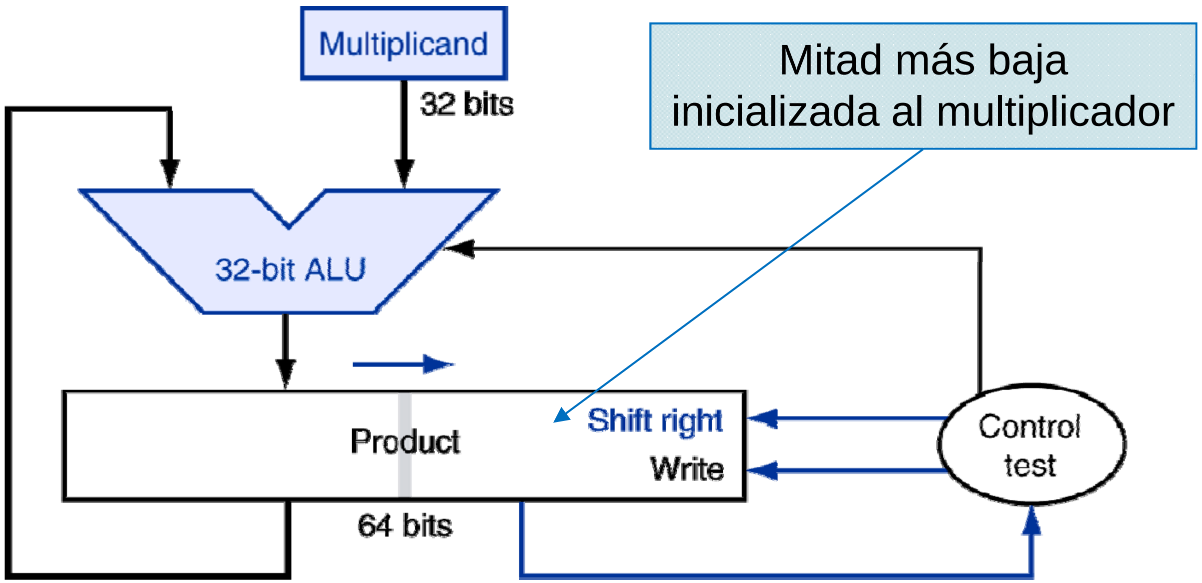




Inicialmente 0

Multiplicación optimizada

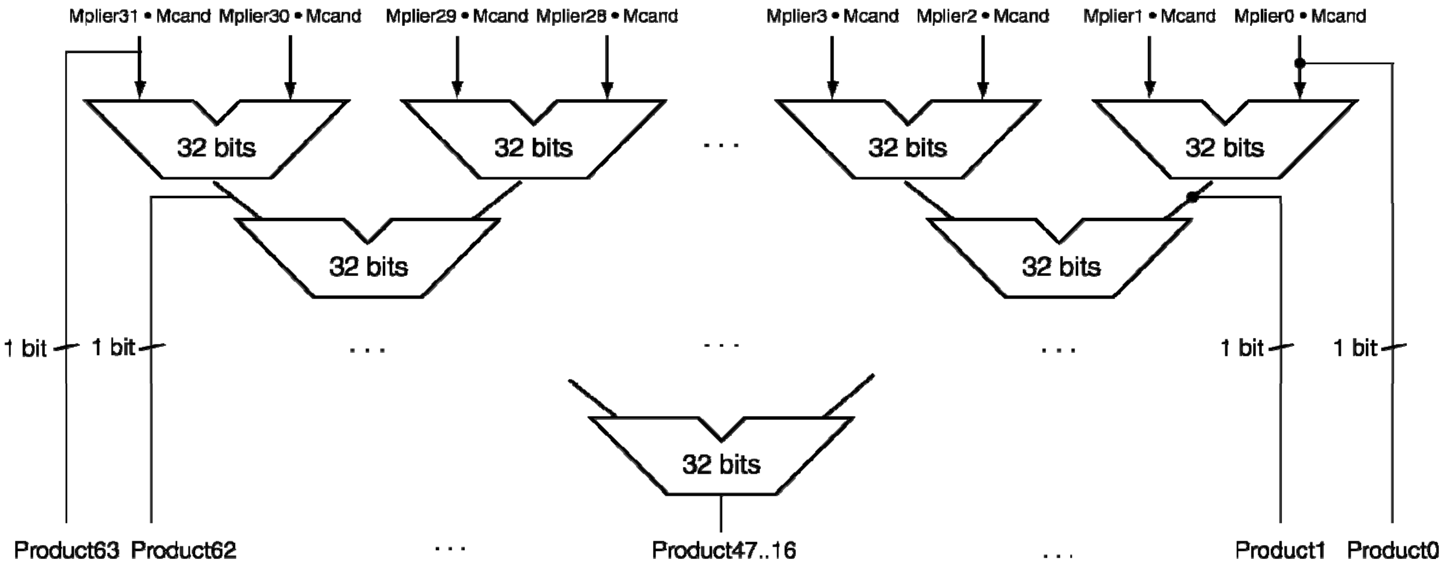
- **Efectúa en parelelo la suma y el desplazamiento**



- Un ciclo por cada suma de producto parcial

Multiplicación rápida

- **Utiliza múltiples sumadores**
  - Buen compromiso entre coste y rendimiento

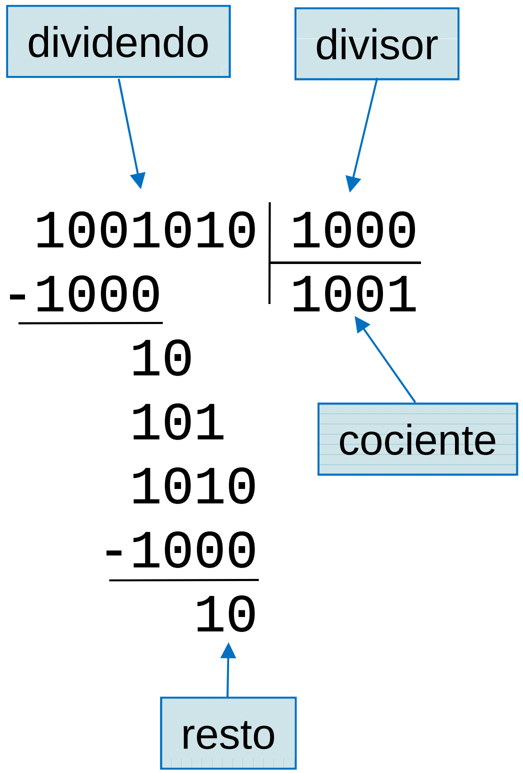


- **Puede efectuarse en un procesador segmentado**
  - Se pueden hecer varias multiplicaciones en paralelo

Multiplicación en MIPS

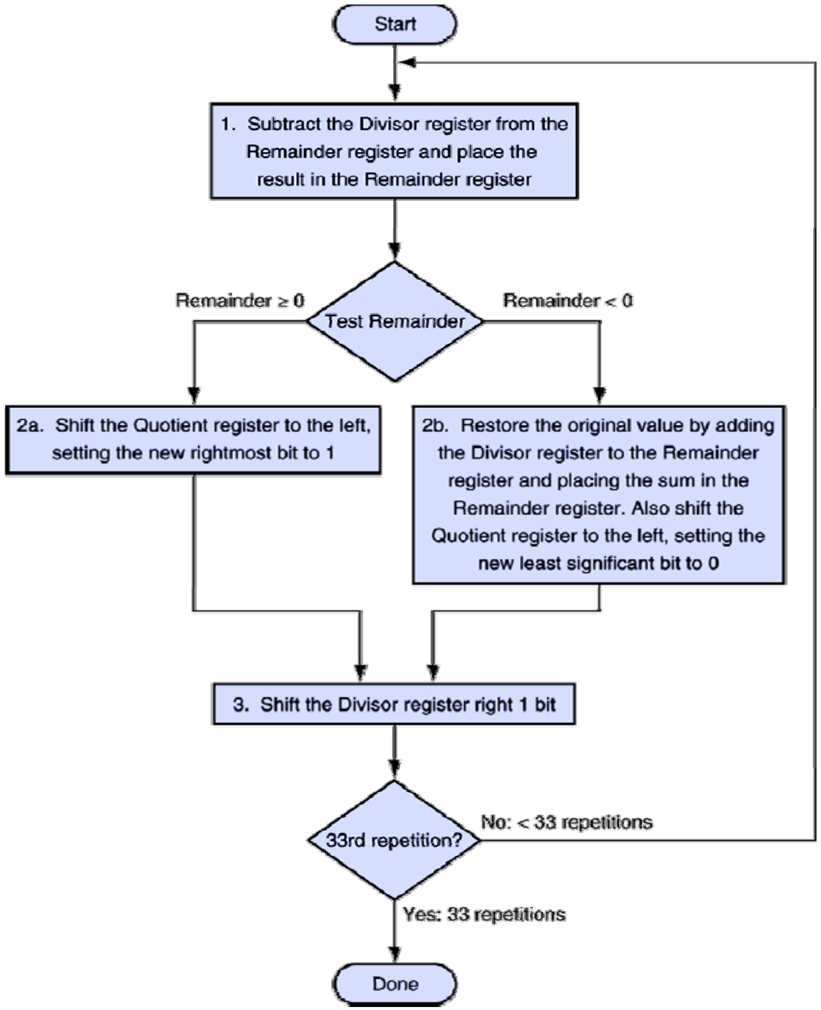
- **MIPS dispone de 2 registros de 32 bits específicos para almacenar el producto (`Hi` y `Lo`)**
  - `Hi`: 32 bits de más peso; `Lo`: 32 bits de menos peso
- **Instrucciones:**
  - `mult rs, rt  /  multu rs, rt`
    - El producto de 64 bits queda en el registro `Hi:Lo`
  - `mfhi rd  /  mflo rd` (*move from* `Lo/Hi`)
    - Mueve el dato de `Hi/Lo` a *rd*
    - Se puede analizar `Hi` para ver si el producto cabe en 32 bits
      - Si `Hi==0` ⇨ El resultado cabe en 32 bits
  - `mul rd, rs, rt` (pseudoinstrucción)
    - Deposita los 32 bits menos significativo del producto en *rd*

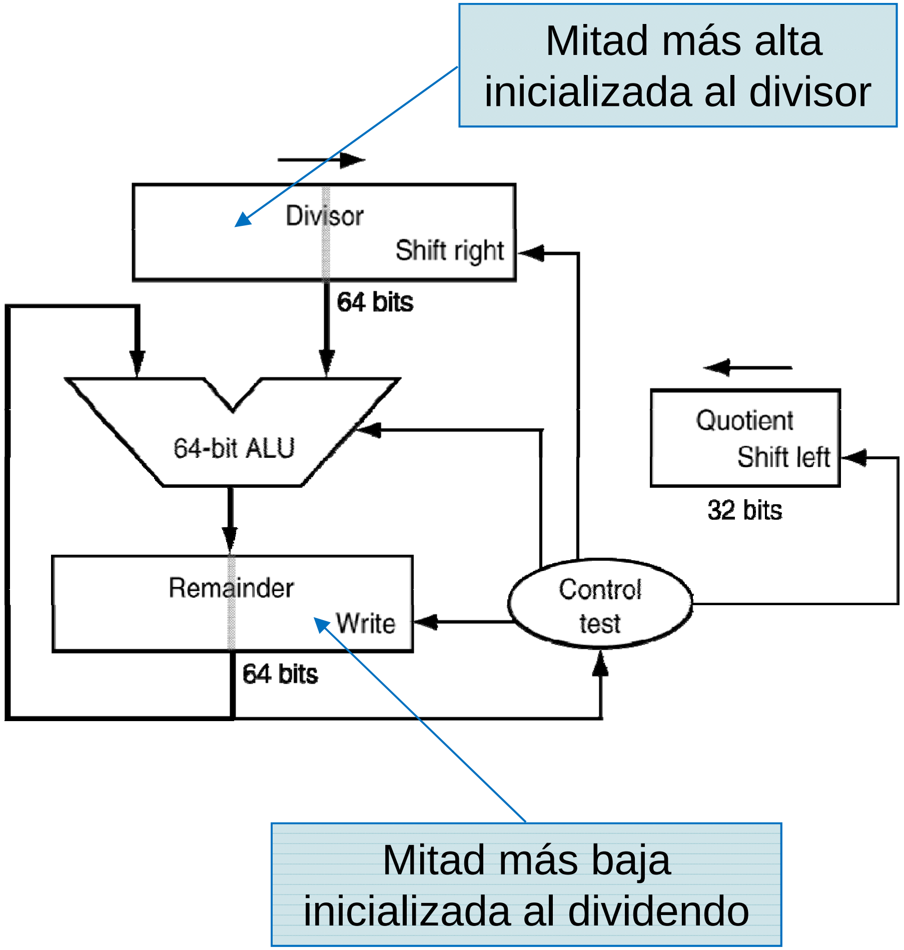
División



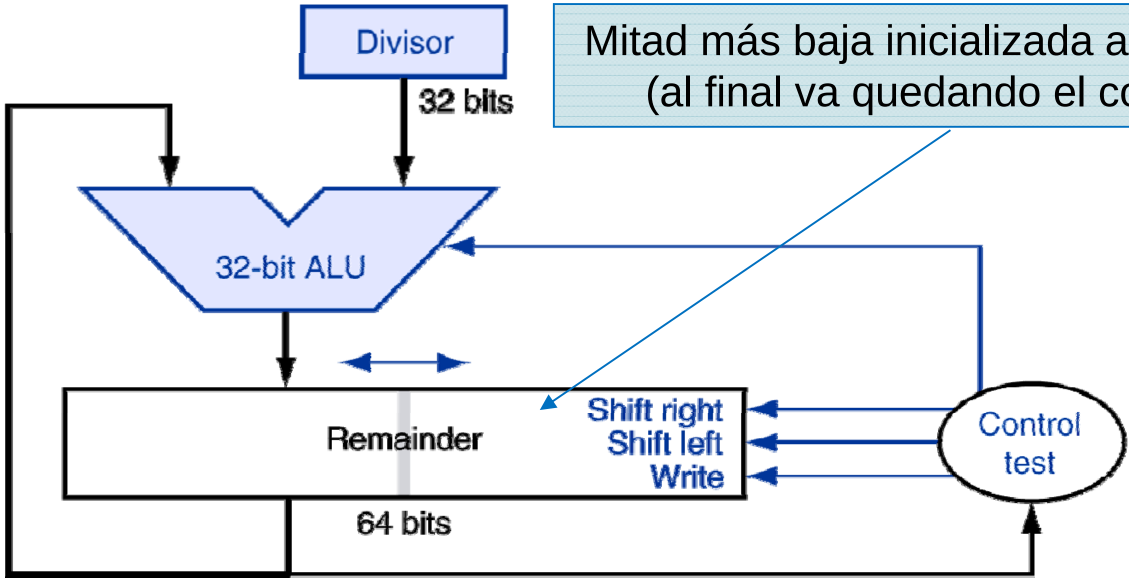
- **Analizar si el divisor es 0: si lo es, FIN**
- **Algoritmo de "lápiz y papel"**
  - Si divisor ≤ dividendo parcial
    - 1 al bit del cociente y se resta el divisor
  - Si no,
    - 0 al bit del cociente y se baja el bit siguiente del dividendo
  - Se repite el proceso hasta acabar con todos los bits del dividendo
- **Division con restauración**
  - Efectuar la resta y si el resto es <0, se suma el divisor para restaurar el dividendo parcial
- **División de enteros con signo**
  - Se dividen los valores absolutos
  - Se ajustan los signos del cociente y resto

División por hardware





Divisor optimizado



Mitad más baja inicializada al dividendo (al final va quedando el cociente)

- **Un ciclo por cada cálculo de resto parcial**
- **Se parece mucho a un multiplicador**
  - Se puede emplear el mismo hardware

División rápida

- **No se puede usar hardware paralelo como en el multiplicador**
  - Porque la resta está condicionada al signo del resto
- **Algunos algoritmos de división rápidos (p.e. algoritmo SRT) generan varios bits del cociente en cada paso**
  - Sin embargo, necesitan varios pasos

División en MIPS

- **Usa los registros `Hi` y `Lo` para el resultado**
  - `Hi`: resto con 32 bits
  - `Lo`: cociente con 32 bits
- **Instrucciones:**
  - `div rs, rt  /  divu rs, rt`
  - No se analiza ni el *overflow* ni la división por 0
    - Si se desea, se puede hacer por software.
  - El resultado se puede recoger mediante las instrucciones `mfhi` y `mflo` (*move from* `Hi/Lo`) (ver transparencia 3.11)

Aritmética de punto flotante

- **Representación de números no enteros**
  - También incluye números muy pequeños y muy grandes
- **Notación similar a la científica**
  - -2.34×10^{56} ← normalizado
  - $+0.002 \times 10^{-4}$ ← no normalizados
  - $+987.02 \times 10^{9}$ ← no normalizados
- **En binario**
  - $\pm 1.xxxx_{(2} \times 2^{yyyy}$

    $1.xxxx_{(2} = mantisa, \quad yyyy = característica$
- **Corresponde a los tipos `float` y `double` de C**

Estándar de punto flotante

- **Definido por el estándar IEEE 754-1985**
- **Desarrollado en respuesta a la divergencia de las representaciones existentes**
  - Garantizar la portabilidad de los programs científicos
- **Actualmente está universalmente aceptado**
- **Dos representaciones:**
  - Simple precisión (32 bits)
  - Doble precisión (64 bits)

Formato IEEE-754 de punto flotante

Precisión: simple: 8 bits, doble: 11 bits — simple: 23 bits, doble: 52 bits

S	Exponente	Fracción

$$x = (-1)^S \times (1.\text{Fracción}) \times 2^{(\text{Exponente}-\text{Exceso})}$$

- **S: bit de signo (0 $\Rightarrow$ no negativo, 1 $\Rightarrow$ negativo)**
- **Normalizar mantisa: 1.0 ≤ |mantisa| < 2.0**
  - Siempre hay un 1 antes del punto; por ello este bit no necesita representarse explícitamente (bit oculto o implícito)
  - La mantisa es el campo “Fracción” con el “1.” restaurado:
    Mantisa=1.Fracción
- **Exponente: representación en exceso: característica + exceso**
  - Representa la característica como un entero sin signo
  - Simple precisión: exceso = 127; Doble precisión: exceso = 1023

Rango y precisión

- **Rango**
  - Conjunto de intervalos donde existen números reales representables
  - El rango está determinado por el número de bits para representar la característica
- **Precisión**
  - Diferencia máxima entre dos números contiguos representables
  - Normalmente se mide de forma relativa, es decir, referida al número representado
  - La precisión está determinada por el número de bits para representar la mantisa

Rango en simple precisión

- **Exponentes `00000000` y `11111111` reservados**
- **Menor valor representable**
  - Exponente: `00000001` $\Rightarrow$ característica = $1 - 127 = -126$
  - Fracción: `000…00` $\Rightarrow$ mantisa = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Mayor valor representable**
  - Exponente: `11111110` $\Rightarrow$ característica = $254 - 127 = 127$
  - Fracción: `111…11` $\Rightarrow$ mantisa $\approx$ (<) 2.0
  - $\approx (<)\ \pm 2.0 \times 2^{+127} = \pm 2^{+128} \approx \pm 3.4 \times 10^{+38}$
- **Intervalo representable:** $(-2^{128}, -2^{-126}] \cup [2^{-126}, 2^{128})$

Rango en doble precisión

- **Exponentes `00000000000` y `11111111111` reservados**
- **Menor valor representable**
  - Exponente: `00000000001` $\Rightarrow$ característica = $1 - 1023 = -1022$
  - Fracción: `000…00` $\Rightarrow$ mantisa = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Mayor valor representable**
  - Exponente: `11111111110` $\Rightarrow$ característica = $2046 - 1023 = 1023$
  - Fracción: 111…11 $\Rightarrow$ mantisa $\approx$ (<) 2.0
  - $\approx (<) \pm 2.0 \times 2^{+1023} = \pm 2^{+1024} \approx \pm 1.8 \times 10^{+308}$
- **Intervalo representable:** $(-2^{1024}, -2^{-1022}] \cup [2^{-1022}, 2^{1024})$

Precisión en punto flotante

- **Precisión relativa**
  - Todos los bits de la mantisa son importantes
  - Simple precisión: aprox. 2^{-23}
    - Para calcular equivalencia en dígitos decimales, calcularemos logaritmos decimales:

      $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 7$ dígitos decimales
  - Doble precisión: aprox. 2^{-52}
    - Equivalente a $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 15$ dígitos decimales

Ejemplo de representación en punto flotante (I)

- **Representar – 0.75(10**
  - $-0.75 = 0.11_2 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fracción = $1000...00_{(2}$ Se ha eliminado el 1 (bit oculto) a 1.1
  - Exponente = característica + exceso = −1 + exceso
    - Simple precisión: $-1 + 127 = 126 = 01111110_{(2}$
    - Doble precisión: $-1 + 1023 = 1022 = 01111111110_{(2}$
- **Simple precisión:** 101111110 1000...00
- **Doble precisión:** 101111111110 1000...00

Signo

Exponente

Fracción

Ejemplo de representación en punto flotante (II)

- **¿Qué número es el representado en simple precisión por la siguiente secuencia de bits?**

  11000000101000…00

  - S = 1
  - Fracción = 01000…00$_{(2}$

    ⇒ Mantisa = 1.Fracción = 1.01
  - Exponente = 10000001$_{(2}$ = 129

    ⇒ Característica = Exponente – Exceso = 129 - 127 = 2

$$x = (-1)^1 \times (1.01_{(2)}) \times 2^{(129-127)} = -1 \times 1.25 \times 2^2 = -5.0$$

Números desnormalizados

- **Exponente = `000...0` y el bit oculto es 0**

$$x = (-1)^S \times (0.\text{Fracción}) \times 2^{-\text{Exceso}}$$

  - Números menores que todos los normales
  - Permiten desbordamiento hacia 0 gradual disminuyendo la precisión
  - Número desnormalizado con fracción = 000...0:

$$x = (-1)^S \times (0.0) \times 2^{-\text{Exceso}} = \pm 0.0$$

Dos representaciones para 0.0

Infinitos y NaN's (*Not a number*)

- **Exponente = 111…1, Fracción = 000…0**
  - ±Infinito
  - Puede usarse en los calculos siguientes, evitando la necesidad de analizar el desbordamiento (*overflow*)
- **Exponente = 111…1, Fracción ≠ 000…0**
  - NaN: No es un número (*Not-a-Number)*
  - Indica resultado indefinido o ilegal
    - Ejemplo: 0.0 / 0.0
  - Los NaN pueden arrastrarse a los cálculos siguientes

Adición en punto flotante (I)

Consideremos un ejemplo con 4 dígitos en decimal

$$9.999 \times 10^1 + 1.610 \times 10^{-1}$$

1. **Alinear puntos decimales (igualar características)**
   - Desplazar el número con menor característica

$$9.999 \times 10^1 + 0.016 \times 10^1$$

2. **Sumar mantisas**

$$9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$$

3. **Normalizar resultado y analizar *over/underflow***

$$1.0015 \times 10^2$$

4. **Redondear y renormalizar si es necesario**

$$1.002 \times 10^2$$

Adición en punto flotante (II)

Consideremos ahora un ejemplo con 4 dígitos binarios

$$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}\ (0.5 + -0.4375)$$

1. **Alinear mantisas (igualar caractarísticas)**
   - Desplazar el número con menor característica

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$$

2. **Sumar mantisas**

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$$

3. **Normalizar el resultado y analizar *over/underflow***

$$1.000_2 \times 2^{-4},\ (\text{no hay } \mathit{over/underflow})$$

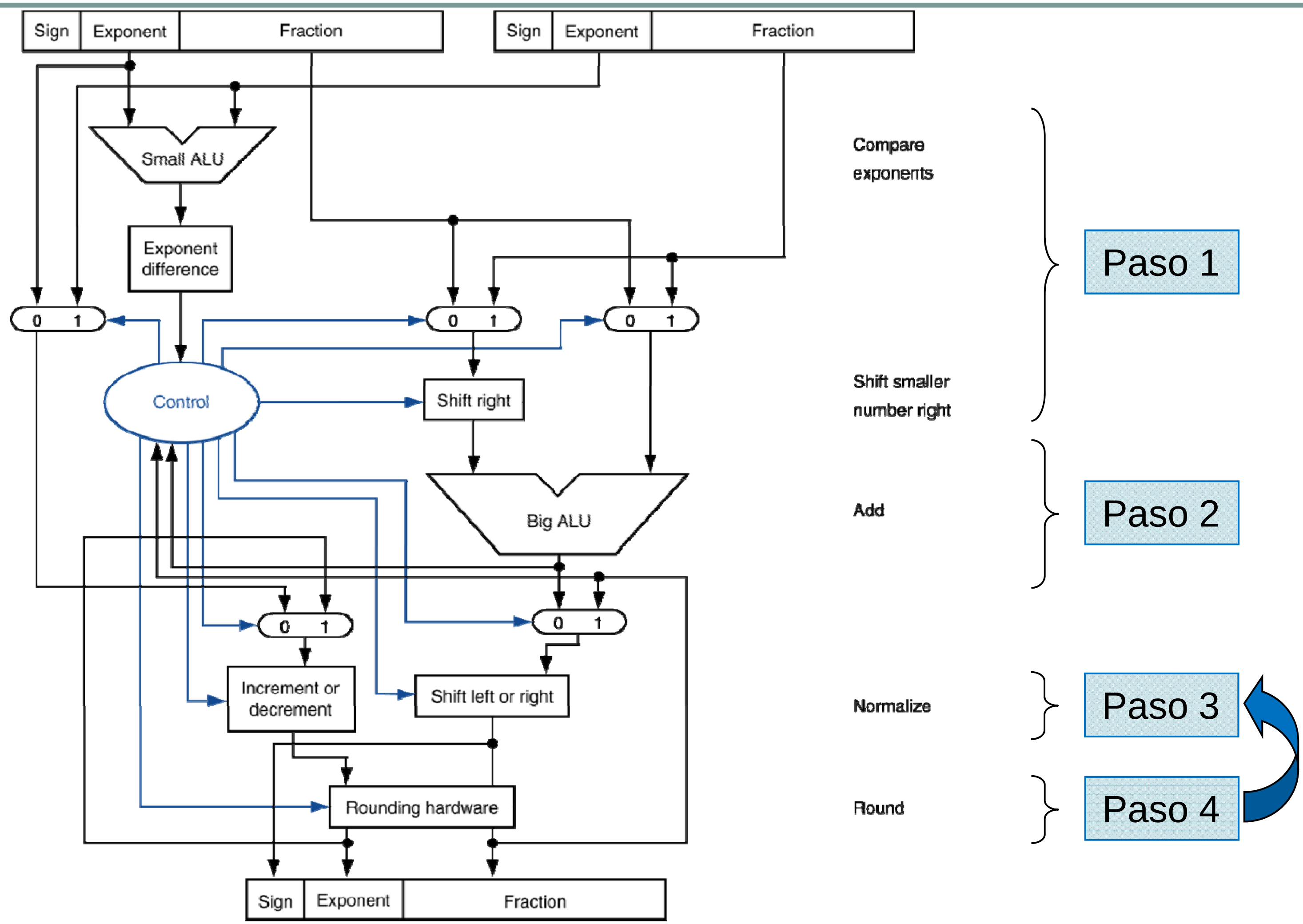
4. **Redondear y renormalizar si es necesario**

$$1.000_2 \times 2^{-4}\ (\text{sin cambios})\ = 0.0625$$

Sumador de punto flotante por hardware (I)

- **Mucho más complicado que un sumador entero**
- **Hacerlo en un solo ciclo haría este muy largo**
  - Sería un ciclo mucho más largo que para las operaciones enteras
  - Un reloj más lento penalizaría todas las instrucciones del procesador
- **Un sumador de punto flotante normalmente emplea varios ciclos**
  - Puede segmentarse

Sumador de punto flotante por hardware (II)



Multiplicación en punto flotante (I)

Consideremos un ejemplo con 4 dígitos en decimal

$$1.110 \times 10^{10} \times 9.200 \times 10^{-5}$$

1. **Sumar exponentes**
   - Para representaciones en exceso, restar el exceso

     Exponente del resultado: $10 + -5 = 5$
2. **Multiplicar las mantisas**

   $$1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$$
3. **Normalizar el resultado y analizar *over/underflow***

   $$1.0212 \times 10^6$$
4. **Redondear y renormalizar si es necesario**

   $$1.021 \times 10^6$$
5. **Determinar el signo: signos distintos ⇔ resultado negativo**

   $$+1.021 \times 10^6$$

Multiplicación en punto flotante (II)

Consideremos ahora un ejemplo con 4 dígitos binarios

$$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}\ (0.5 \times -0.4375)$$

1. **Sumar exponentes y restar el exceso si lo hay**
   - Sin exceso: $-1 + -2 = -3$
   - Con exceso: $(-1{+}127) + (-2{+}127) - 127 = -3 + 254 - 127 = -3{+}127$
2. **Multipicar las mantisas**

   $$1.000_2 \times 1.110_2 = 1.1102 \;\Rightarrow\; 1.110_2 \times 2^{-3}$$
3. **Normalizar el resultado y analizar *over/underflow***

   $$1.110_2 \times 2^{-3}\ (\text{sin cambios y sin } over/underflow)$$
4. **Redondear y renormalizar si es necesario**

   $$1.110_2 \times 2^{-3}\ (\text{sin cambios})$$
5. **Determinar el signo: signos distintos ⇔ resultado negativo**

   $$-1.110_2 \times 2^{-3} = -0.21875$$

Hardware para la aritmética en punto flotante

- **La complejidad de un multiplicador de punto flotante es similar a la de un sumador**
  - Utiliza un multiplicador para las mantisas en vez de un sumador
- **El harware de punto flotante normalmente efectúa:**
  - Adición, sustracción, multiplicación, división, cálculo del recíproco y de la raíz cuadrada
  - Conversión entre entero y punto flotante y viceversa
- **Las operaciones necesitan varios ciclos**
  - Pueden segmentarse

Instrucciones de punto flotante en MIPS (I)

- **El hardware de PF es el coprocesador 1**
  - Procesador adjunto que extiende la ISA
- **Registros separados de punto flotante**
  - 32 registros de simple precisión: `$f0`, `$f1`, … `$f31`
  - Pares de registros de doble precisión: `$f0/$f1`,`$f2/$f3`,…
    - La release 2 de la ISA MIPS posee 32 registros de PF de 64 bits
- **Instrucciones de PF solo operan con los registros de PF**
  - Generalmente los programas no efectúan operaciones enteras sobre datos de PF o viceversa
  - Más registros sin impacto sobre el tamaño del código
- **Instrucciones de carga/almacenamiento para FP**
  - `lwc1`, `ldc1`, `swc1`, `sdc1` (equivalentes a `l.s`, `l.d`, `s.s` y `s.d`)
    - ej., `ldc1 $f8, 32($sp)`

Instrucciones de punto flotante en MIPS (II)

- **Aritmética de PF con simple precisión**
  - `add.s`, `sub.s`, `mul.s`, `div.s`
    - ej., `add.s $f0, $f1, $f6`
- **Aritmética de PF con doble precisión**
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - ej., `mul.d $f4, $f4, $f6`
- **Comparación en simple y doble precisión**
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* es la condición: `eq`, `lt`, `le`, ...)
  - Activa o desactiva los bits de condición de PF
    - ej.: `c.lt.s $f3, $f4`
- **Bifurcación en función de bits de condición de PF**
  - `bc1t`, `bc1f`
    - ej., `bc1t Destino`

Ejemplo 1 de cálculo en punto flotante: conversión de °F a °C

- $C = 5/9(F - 32)$
- **Código de alto nivel:**

```
float f2c (float fahr)
    {return ((5.0/9.0)*(fahr - 32.0));}
```

  - `fahr` en \$f12, resultado en \$f0, las constantes están en el área global de memoria

- **Código MIPS:**

```
f2c: lwc1  $f16, const5($gp)
     lwc1  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

Ejemplo 2 de cálculo en punto flotante: multiplicación de matrices (I)

- **X = X + Y × Z**
  - **X**, **Y** y **Z**: Matrices de 32 × 32 elementos de doble precisión en 64 bits
- **Código de alto nivel (C):**

```
void mm (double x[][],double y[][],double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Direcciones de `x`, `y` y `z` en `$a0`, `$a1` y `$a2`
  - `i`, `j` y `k` en `$s0`, `$s1` y `$s2`

Ejemplo 2 de cálculo en punto flotante: multiplicación de matrices (II)

Código MIPS:

```
      li   $t1, 32         # $t1 = 32 (row size/loop end)
      li   $s0, 0          # i = 0; initialize 1st for loop
L1:   li   $s1, 0          # j = 0; restart 2nd for loop
L2:   li   $s2, 0          # k = 0; restart 3rd for loop
      sll  $t2, $s0, 5     # $t2 = i * 32 (size of row of x)
      addu $t2, $t2, $s1   # $t2 = i * size(row) + j
      sll  $t2, $t2, 3     # $t2 = byte offset of [i][j]
      addu $t2, $a0, $t2   # $t2 = byte address of x[i][j]
      l.d  $f4, 0($t2)     # $f4 = 8 bytes of x[i][j]
L3:   sll  $t0, $s2, 5     # $t0 = k * 32 (size of row of z)
      addu $t0, $t0, $s1   # $t0 = k * size(row) + j
      sll  $t0, $t0, 3     # $t0 = byte offset of [k][j]
      addu $t0, $a2, $t0   # $t0 = byte address of z[k][j]
      l.d  $f16, 0($t0)    # $f16 = 8 bytes of z[k][j]
      …
```

Ejemplo 2 de cálculo en punto flotante: multiplicación de matrices (III)

…

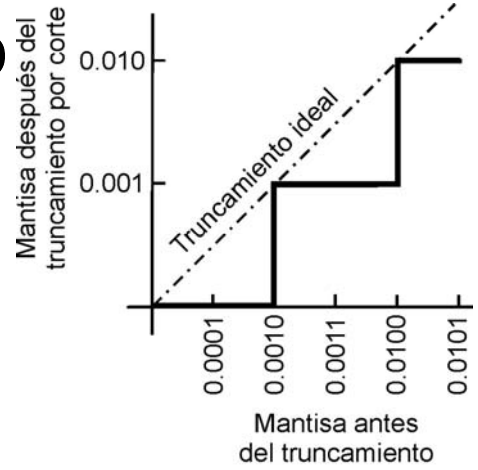
```
sll  $t0, $s0, 5        # $t0 = i*32 (size of row of y)
addu  $t0, $t0, $s2     # $t0 = i*size(row) + k
sll   $t0, $t0, 3       # $t0 = byte offset of [i][k]
addu  $t0, $a1, $t0     # $t0 = byte address of y[i][k]
l.d   $f18, 0($t0)      # $f18 = 8 bytes of y[i][k]
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16    # f4=x[i][j] + y[i][k]*z[k][j]
addiu $s2, $s2, 1       # $k k + 1
bne   $s2, $t1, L3      # if (k != 32) go to L3
s.d   $f4, 0($t2)       # x[i][j] = $f4
addiu $s1, $s1, 1       # $j = j + 1
bne   $s1, $t1, L2      # if (j != 32) go to L2
addiu $s0, $s0, 1       # $i = i + 1
bne   $s0, $t1, L1      # if (i != 32) go to L1
```

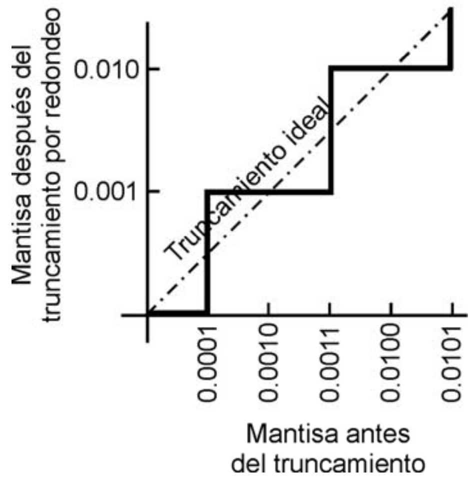
Aritmética de precisión (I)

- **Los procesadores de PF utilizan algunos bits extras para realizar los cálculos (bits de reserva)**
- **Los bits de reserva se alimentan de**
  - Las llevadas en las sumas
  - Los bits que salen por la derecha en los desplazamientos
- **El proceso para eliminar los bits de reserva del resultado final se llama truncamiento**
- **Medida del error en el truncamiento**
  - ulp (*unit in de last place*: último bit conservado)

Aritmética de precisión (II)

- **Métodos básicos de truncamiento**
  - Truncamiento por corte
    - Se eliminan los bits adicionales
    - Error: 0 a -1 ulp
  - Truncamiento por redondeo
    - Se suma el bit de reserva de más orden al menos significativo de los conservados
    - Error: de -1/2 ulp a +1/2 ulp





Aritmética de precisión (III)

- **La norma IEEE-754 respecto al redondeo**
  - Bits de reserva: guarda, redondeo y adherente (*sticky bit*)
  - Se puede escoger el método de truncamiento entre
    - Al más próximo (truncamiento por redondeo)
    - Hacia +∞
    - Hacia -∞
    - Hacia 0 (truncamiento por corte)
- **No todas las unidades de PF implementan todas las opciones**
  - La mayoría de los lenguajes de programación y bibliotecas de PF solo utilizan las opciones de defecto
- **Equilibrio entre la complejidad del hardware, el rendimiento y las necesidades del mercado**

Interpretación de los datos

Recuadro importante

- **Los bits no tienen significado por sí mismos**
  - Su interpretación depende del contexto y de la instrucción aplicada
- **Representación de números en computadores**
  - Rango y precisión finitos
  - Es necesario tenerlo en cuenta en los programas

Conclusiones

- **Las ISAs admiten instrucciones aritméticas**
  - Enteros con signo y sin signo
  - Representación en punto flotante: aproximación a los reales
- **Rango y precisión limitados**
  - Las operaciones pueden provocar *overflow* y *underflow* (desbordamiento y subdesbordamiento)
- **MIPS ISA**
  - Las instrucciones básicas son 54 (las más frecuentes)
  - Otras instrucciones menos frecuentes no se implementan