



**DCC**  
DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

**IIC2343**

# **Arquitectura de Computadores**

**Clase 6 - Representaciones Numéricas II**

**Profesor: Germán Leandro Contreras Sagredo**

## Objetivos de la clase

- Entender el paso de base decimal a binaria para números racionales.
- Conocer distintas representaciones de números racionales usando bits, entendiendo sus ventajas y desventajas.
- Realizar ejercicios que consoliden los conocimientos anteriores.

## Hasta ahora...

Ya sabemos representar números enteros en base binaria a través del complemento de 2 y la forma de operar con ellos en el computador básico apoyándonos con la lógica booleana.

Ahora, ¿qué pasa con los números racionales?

- En el contexto del computador básico de este curso, no se utilizan.
- En la práctica, representar y operar estos números es esencial en los procesadores.

## Representaciones de números racionales

¿Cómo representamos los números racionales de forma posicional?

**Realizando la expansión con exponentes negativos.**

**Ejemplo:**  $123,45 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0 + 4 * 10^{-1} + 5 * 10^{-2}$

Podemos hacer lo mismo en base binaria.

**Ejemplo:**  $101,01b = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} = 5,25$

Ahora, ¿cómo pasamos de base decimal a binaria?

## Representaciones de números racionales - División binaria

Podemos obtener el valor en binario de un número racional en base decimal **obteniendo el valor de la división binaria**.

**Ejemplo:**  $0,75 = 3 / 4 = 11b / 100b$

En este caso, la división se realiza igual que la división decimal tradicional, pero realizando **operaciones aritméticas binarias**.

## Representaciones de números racionales - División binaria

Veremos a continuación la división binaria del ejemplo anterior.

$$11 : 100 = 0.$$

11(0)

100 no “cabe” en 11, por lo que preliminarmente el valor es 0. Agregamos un 0 al resto (110) y volvemos a operar para obtener los decimales.

$$110 : 100 = 0.1$$

-100

100 “cabe” una vez en 110, por lo que agregamos el primer decimal 1. Agregamos un 0 al resto (0100 = 100) y volvemos a dividir.

010(0)

$$100 : 100 = 0.11$$

000

Al ser el resto igual al divisor, agregamos un último decimal 1 al resultado y termina la división.

**Finalmente:**  $0,75 = 0.11_b = 1 * 2^{-1} + 1 * 2^{-2}$

## Representaciones de números racionales - División binaria

Ahora... ¿Cómo obtenemos el valor 0.1 en binario?

$$10^{-1} = 0,1 = 1 / 10 = 1b / 1010b$$

$$1 : 1010 = \mathbf{0.000}$$
  

$$1(\mathbf{0000})$$

1010 no “cabe” en 1, por lo que preliminarmente el valor es 0. Agregamos 0's al resto hasta que el divisor “quepa” en él y volvemos a operar.

$$\begin{array}{r} \text{---} \text{---} \text{---} \text{---} \text{---} \text{---} \\ 10000 : 1010 = 0.0001 \\ -01010 \\ \mathbf{00110(0)} \end{array}$$

1010 “cabe” una vez en 10000, por lo que agregamos el decimal 1. Agregamos un 0 al resto (001100 = 1100) y volvemos a dividir.

$$\begin{array}{r} \text{---} \text{---} \text{---} \text{---} \text{---} \text{---} \\ 1100 : 1010 = 0.00011 \\ -1010 \\ \mathbf{0010(0)} \end{array}$$

1010 “cabe” una vez en 1100, por lo que agregamos el decimal 1. Agregamos un 0 al resto (10 = 100) y notamos que en la próxima iteración necesitaremos añadir nuevamente dos 0's para volver a operar. ¿Qué ocurre en este caso?

## Representaciones de números racionales - División binaria

$$100 : 1010 = 0.0001100$$

100(00)

1010 no “cabe” en 100, por lo que agregamos 0’s al resultado y resto hasta que el divisor “quepa” en él y volvemos a operar.

$$10000 : 1010 = 0.00011001$$

-01010

00110(0)

1010 “cabe” una vez en 10000, por lo que agregamos el decimal 1. Nos damos cuenta que esta división ya la hicimos previamente.

$$1100 : 1010 = 0.000110011$$

-1010

0010(0)

Si seguimos dividiendo, podemos notar que empezaremos a repetir la operación de división hasta el infinito, generando la misma secuencia de decimales “0011”.

De esta forma, nos damos cuenta que:

$$(0.1)_{10} = 0.0\overline{0011}b$$



## Representaciones de números racionales - Infinitos

¿Significa esto que números finitos en base decimal pueden ser infinitos en base binaria?

**Sí**, y vice-versa. Por ejemplo, tomemos el caso  $0.\overline{3}$  en base ternaria.

$$(0.\overline{3})_{10} = 1 / 3 = (1)_3 / (10)_3 = (0.1)_3$$

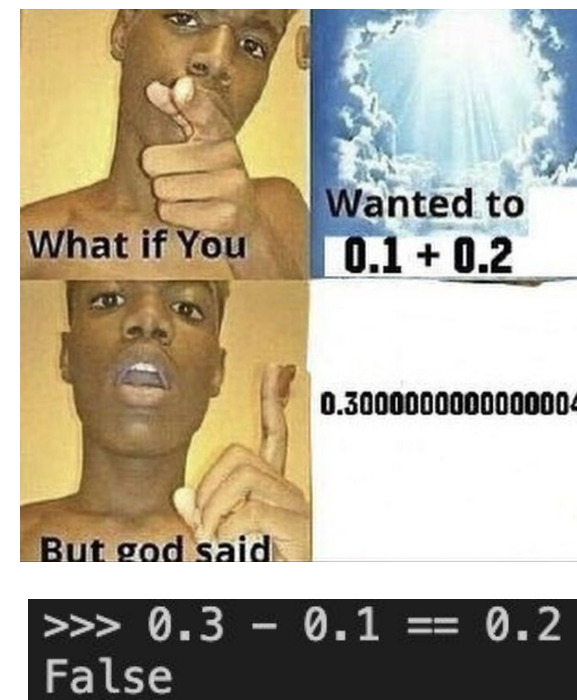
En resumen, la base escogida puede afectar en la finitud del número racional que se busca representar.

## Representaciones de números racionales - Infinitos

Entonces, ¿un computador no puede representar el valor exacto de 0.1?

Sí.

Al operar con el decimal 0.1, no toda operación dará el valor esperado por lo que se conoce como error de redondeo de punto flotante, proveniente de la limitante física para representar números infinitos en base binaria.



Ejemplo en Python 3.9.

## Representaciones de números racionales - Caso real

**Misil Patriot (1991):** Sistema que intercepta objetivos aéreos con misiles. No interceptó un misil Scud (Unión Soviética) por una falla en el seguimiento del objetivo por la aproximación de un decimal finito mediante un número binario infinito. Fallecieron 28 personas.

- El algoritmo de seguimiento medía el tiempo en 1/10 seg. y era almacenado en un registro de 24 bits.
- El error por el redondeo de estos registros es de  $9.5 * 10^{-8}$  seg. Después de 100 horas, el error acumulado fue de 0.34 seg.
- El misil Scud viaja a 1.7km/s. Alcanzó a recorrer más de medio kilómetro en los 0.34 segundos de error del sistema.

## Representaciones de números racionales

Ahora que entendemos mejor cómo se ven los números racionales en base decimal, veremos dos tipos de representación distintas:

- Representación de punto fijo.
- Representación de punto flotante.

## Representaciones de números racionales - Punto fijo

Dados  $N$  dígitos (bits), estos se dividen de manera **fija** para representar signo, parte entera y parte decimal. Se llama punto fijo (*fixed point*) ya que por su formato el punto decimal queda en la misma posición.

### Ejemplo

$N = 8$  bits = 1 bit signo (s) + 3 bits parte entera (t) + 4 bits fracción (f)

$$10,110b = \underset{s}{0}\underset{t}{010}\underset{f}{0110} \quad -1,0011b = \underset{s}{1}\underset{t}{001}\underset{f}{0011}$$

## Representaciones de números racionales - Punto fijo

### Ventajas

- Simple y rápido de ocupar.

### Desventajas

- Rango **muy** pequeño. En el ejemplo anterior, el número representable más grande es **01111111** = 7,875. Con números enteros de 8 bits, este número es 127.

Podemos mejorar esto haciendo que el punto decimal sea *dinámico*.

## Representaciones de números racionales - Punto flotante

Se basa en la notación científica normalizada: representación del número como la multiplicación entre un número (el significante) y una base elevada a un exponente.

### Ejemplo

$$(124)_{10} = (1.24)_{10} * 10^2$$

¿Y en base binaria?

$$01111100b = 1.111100b * 10^{110}$$

El punto decimal “flota” tantos bits como lo indique el exponente.

En este caso, el exponente flota hacia la derecha 6 posiciones.

## Representaciones de números racionales - Punto flotante

Dados  $N$  dígitos (bits), los dividimos en un **significante** (o mantisa) y un **exponente**, ambos con signo.

### Ejemplo

$N = 8$  bits = 1 bit signo significante (ss) + 3 bits significante (s) +  
1 bit signo exponente (se) + 3 bits exponente (e)

$$10,110b = 1,011 * 10^{001} = \underset{ss}{0} \underset{s}{101} \underset{se}{0} \underset{e}{001}$$

Perdemos los 2 bits menos significativos por redondeo

$$-0,00011b = 0,11 * 10^{-011} = \underset{ss}{1} \underset{s}{011} \underset{se}{1} \underset{e}{011}$$



# Representaciones de números racionales - Punto flotante

## Ventajas

- Rango mucho más amplio de valores.

## Desventajas

- Pérdida de precisión. Ganamos mucho rango con un bit de exponente, pero perdemos precisión en los bits de significante.

A pesar de la desventaja, se acepta este *trade-off* ya que se puede solventar con más bits en la representación.

## Representaciones de números racionales - Punto flotante

### *Machine epsilon* o *epsilon* de la máquina

Número utilizado para medir la distancia entre dos números consecutivos bajo una representación de punto flotante. Sirve de referencia para medir la **pérdida de precisión** al truncar un número por la cantidad limitada de bits del significante.

Formalmente, es la **distancia entre el valor 1 y el número más pequeño representable siguiente al 1**.

## Representaciones de números racionales - Punto flotante

### ***Machine epsilon o epsilon de la máquina***

Bajo esta representación:

$$1 = 0_{ss} 100_s 0_{se} 000_e = 1,00 * 10^0 = 1$$

$$next(1) = 0_{ss} 101_s 0_{se} 000_e = 1,01 * 10^0 = 1.01$$

$$\epsilon_{mach} = 1 - next(1) = 0.01 = 10^{-2}$$

## Representaciones de números racionales - Punto flotante

### ***Machine epsilon o epsilon de la máquina***

Este valor supone una cuota **superior** respecto al error absoluto que tenemos con el truncado, esto es:

$$\left| \frac{x - y}{x} \right| \leq \epsilon_{\text{mach}}$$

Siendo ***x*** el valor real de nuestro número e ***y*** el valor que adquiere en nuestra representación.

## Representaciones de números racionales - Punto flotante

### ***Machine epsilon o epsilon de la máquina***

Tomemos como ejemplo el número 1111b y veamos su error de representación:

$$x = 1111b$$

$$y = 0_{ss} 111_s 0_{se} 010_e = 111b$$

$$\left| \frac{1111 - 0111}{1111} \right| = \frac{0001}{1111} = 0.\overline{0001} \leq 0,01$$

## Representación de números de punto flotante - IEEE754

Actualmente, la representación de números de punto flotante más utilizada es la indicada por el **estándar IEEE754**, definida en 1985 por el *Institute of Electrical and Electronics Engineers* (quienes también estandarizaron el lenguaje VHDL).

Definen dos tipos de número:

- **Float:** 32 bits.
- **Double:** 64 bits.



## Representación de números de punto flotante - IEEE754

### Float (*Single Precision Floating Point*)

- Signo: 1 bit.
- Exponente **desfasado**: 8 bits. El exponente se desfasa en 127 para no tener que depender de un bit de signo ni usar representación de complemento de 2.
- Significante **normalizado**: 23 bits. Tiene precisión de 24 bits porque **asume que todo significativo parte en 1** (i.e. 1.xxx), por ende, se desprende del uso de dicho bit.

## Representación de números de punto flotante - IEEE754

### Float (*Single Precision Floating Point*)

- Dado que se asume que el significante posee un 1 implícito a la izquierda del punto decimal, no hay forma directa de representar el cero. Por estándar, entonces, se reserva el exponente 00000000 exclusivamente para representar este número, en conjunto con el significante 000000000000000000000000.
- Esto implica que habrá dos representaciones para el cero:  
 $0 = 00000000000000000000000000000000$   
 $0 = 10000000000000000000000000000000$



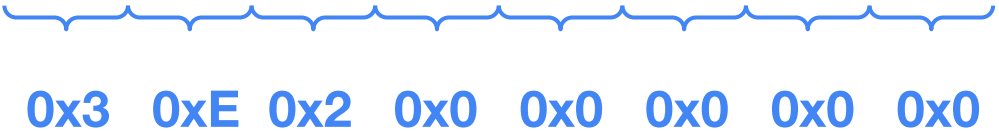
## Representación de números de punto flotante - IEEE754

### Float (*Single Precision Floating Point*)

- También se reserva el exponente 11111111 para estos casos:
  - +Infinity = 01111111100000000000000000000000
  - -Infinity = 11111111100000000000000000000000
  - NaN = 011111111xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
- En el caso de NaN (**Not A Number**), se debe cumplir con que al menos un bit X sea igual a 1.
- Podemos usar [este simulador](#) para ver el formato de un número x.

## Representación de números de punto flotante - IEEE754

### Float (*Single Precision Floating Point*)

- Podemos escribir el número en formato hexadecimal.
- **Ejemplo:**  $00111110001000000000000000000000 = 0x3E200000$   
  
 $0x3 \quad 0xE \quad 0x2 \quad 0x0 \quad 0x0 \quad 0x0 \quad 0x0 \quad 0x0$
- Usando el algoritmo visto en la clase 1, podemos obtener ambos formatos rápidamente.

## Representación de números de punto flotante - IEEE754

### Float (*Single Precision Floating Point*)

- En este caso,  $\epsilon_{\text{mach}}$  corresponde a  $10^{-23}$ .
- Se puede ver que, en el estándar IEEE754, el *epsilon* de la máquina va en función de los bits de significante:

$$\epsilon_{\text{mach}} = 10^{-(\text{bits de significante})}$$

## Representación de números de punto flotante - IEEE754

Un número de punto flotante, en estándar IEEE754, se ve así:

$$X = (-1)^{\text{signo}} * 1.\text{significante} * 10^{(\text{exponente} - 127)\text{b}}$$

**Ejemplo:**  $0.00101\text{b} = (1.01 * 10^{-11})\text{b} = 1 * 2^{-3} + 1 * 2^{-5} = 0.15625$

En este caso,  $\text{exp} - 127 = -3 \rightarrow \text{exp} = 124 = 01111100\text{b}$

Siguiendo el estándar:  $0.00101\text{b} = (-1)^0 * 1.\text{01} * 10^{01111100}$

Finalmente:  $0.00101\text{b} = 00111110001000000000000000000000$  IEEE754

## Representación de números de punto flotante - IEEE754

### Double (*Double Precision Floating Point*)

- Signo: 1 bit.
- Exponente **desfasado**: 11 bits. El exponente se desfasa en 1023.
- Significante **normalizado**: 52 bits.
- Su  $\epsilon_{\text{mach}}$  es igual a  $10^{-52}$ .

## Representación de números de punto flotante - Más opciones

### Decimales como números enteros

- Se utiliza como unidad el **menor valor decimal requerido**.
- **Ejemplo:** Trabajar solo en milisegundos.  $103.1 \text{ seg.} = 103100 \text{ ms.}$
- **Ventaja:** Fácil de operar.
- **Desventaja:** Pérdida de rango (no tenemos exponente).

# Representación de números de punto flotante - Más opciones

## Punto flotante con base decimal

- Se cambia la base 2 por base 10.
- **Ventaja:** Más intuitivo para el humano y elimina problemas de aproximación binaria (0.1, por ejemplo).
- **Desventaja:** Mucho más lento (conversión a nivel de *software* entre base binaria y decimal).

Ejemplo en Python 3.9.

```
>>> from decimal import Decimal
>>> Decimal('0.3') - Decimal('0.1') == Decimal('0.2')
True
```

## Representación de números de punto flotante

### Punto flotante con base decimal y precisión arbitraria

- Base 2 por base 10, pero el tamaño asignado al significante y exponente aumentan según necesidad.
- **Ventaja:** Versión más amplia y precisa para operar con números racionales con gran cantidad de decimales.
- **Desventaja:** Tiempo de procesamiento aún mayor.

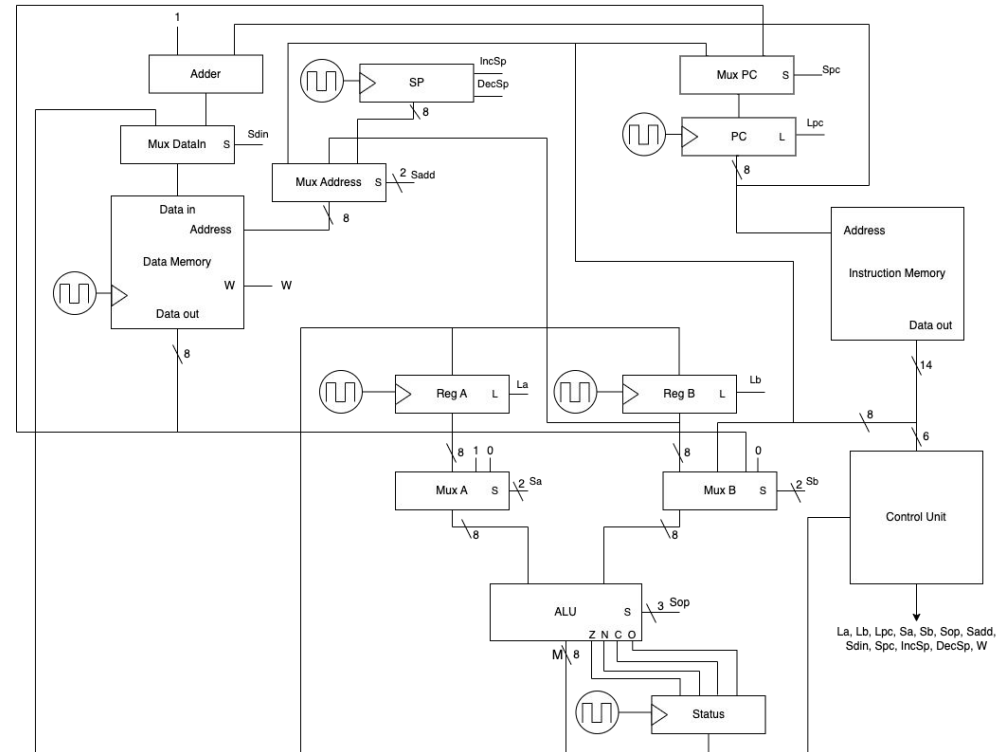
`decimal.Decimal` en Python equivale a esta representación.



## Computador básico - Punto flotante

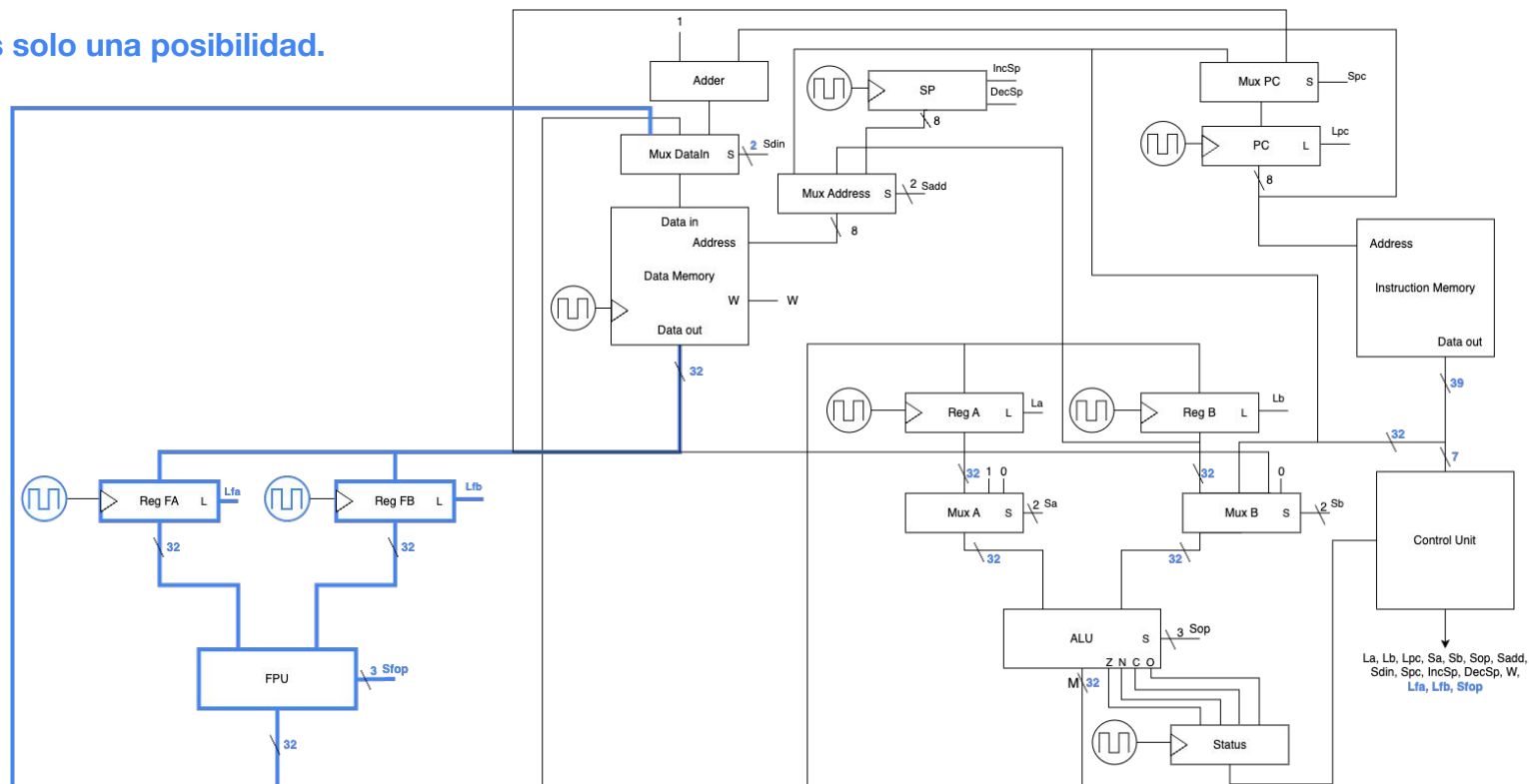
¿Cómo podríamos modificar el diagrama de nuestro computador para soportar números de punto flotante?

No lo usaremos en el contexto de este curso, pero podemos realizar una posible implementación.



# Computador básico - Punto flotante

Esta es solo una posibilidad.



## Computador básico - Punto flotante

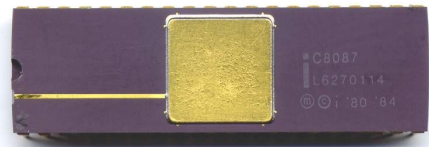
- Aumento de buses de datos, registros y literal a 32 bits.
- Direcciones o muxes de 8 bits pueden truncarse para utilizar solo los 8 bits menos significativos.
- Se añaden nuevos registros *FA*, *FB* para almacenar números que seguirán el estándar IEEE754.
- Para operar sobre *floats*, añadimos una FPU (***F*loating *P*oint *U*nit**). Permiten realizar operaciones como suma, resta, multiplicación, división, logaritmo, raíces, funciones trigonométricas, etc.

## Computador básico - Punto flotante

Las FPU funcionan como un co-procesador, una unidad separada con la que se comunica la CPU para operaciones de punto flotante.



Procesador i8088 (CPU).



Co-procesador i8087 (FPU).  
Se comunica con i8088.

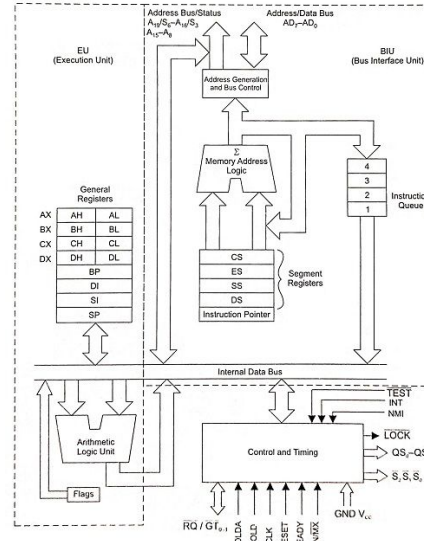
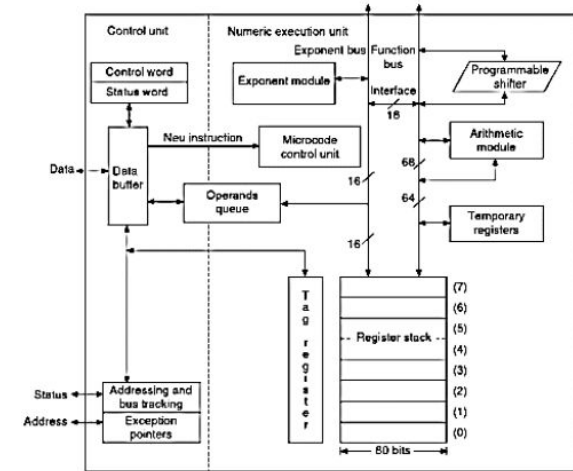


Fig. 5.24 Block diagram of 8088 microprocessor



Block Diagram of 8087 math coprocessor

Diagrama de bloques de ambos procesadores.

## Ejercicios

Ahora, veremos algunos ejercicios.

Estos se basan en preguntas de tareas y pruebas de semestres anteriores, por lo que nos servirán de preparación para las evaluaciones.



## Ejercicios

Se tienen dos números de punto flotante de precisión simple en formato IEEE754:  $A = 0x3E200000$  y  $B = 0x00000000$ . ¿Cuál es el resultado, en formato IEEE754, de  $A : B$ ?

## Ejercicios

Explique por qué el número  $2^{50} + 5$  no puede representarse de manera exacta usando el tipo de dato *float* de 32 bits.

## Ejercicios

Demuestre que los números de punto flotante del tipo *float* del estándar IEEE754 **no** cumplen el principio de asociatividad de la suma, *i.e.*,  $x + (y + z) = (x + y) + z$



## Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?





**DCC**  
DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

**IIC2343**

# **Arquitectura de Computadores**

**Clase 6 - Representaciones Numéricas II**

**Profesor: Germán Leandro Contreras Sagredo**

## Anexo - Resolución de ejercicios

### ¡Importante!

Estos ejercicios pueden tener más de un desarrollo correcto. Las respuestas a continuación no son más que soluciones que **no excluyen** otras alternativas igual de correctas.



## Ejercicios - Respuesta

Se tienen dos números de punto flotante de precisión simple en formato IEEE754:  $A = 0x3E200000$  y  $B = 0x00000000$ . ¿Cuál es el resultado, en formato IEEE754, de  $A : B$ ?

Aquí, la gracia es ver que bajo el estándar IEEE754, la división por el 0 entrega un número infinito. Antes de anotar el resultado, necesitamos saber si el número es positivo, lo que vemos a partir de A:

$0x3E200000 = 00111111000100000000000000000000$

Como este estándar nos dice que el primer bit indica el signo, sabemos que A es positivo. Finalmente, la representación de un infinito positivo en IEEE754 es:

$0x7F800000 = 01111111100000000000000000000000$

## Ejercicios - Respuesta

Explique por qué el número  $2^{50} + 5$  no puede representarse de manera exacta usando el tipo de dato *float* de 32 bits.

Es necesario recordar la composición de un `float` de 32 bits, en el orden dado (de izquierda a derecha):

- **Signo:** 1 bit.
- **Exponente:** 8 bits.
- **Significante:** 23 bits.

Recordando que el exponente está desplazado en 127 unidades, tenemos que  $2^{50}$  se expresa de la siguiente forma:

- **Signo:** 0 (positivo).
- **Exponente:** 10110001 (177).
- **Significante:** 0000000000000000000000.

Ahora, es importante notar que si se le suman 5 unidades al número, estas se deben ver reflejadas en el significante. Esto corresponde a un 1 que se encuentra 50 posiciones a la izquierda (proveniente de  $2^{50}$ ), y un 101 al final (proveniente de 5). Como la cantidad de ceros entre el 1 y el 101 supera el número de bits a disposición para el significante,  $2^{50} + 5$  no se puede representar.

**Nota:** Esto pasa para todos los números de la forma  $2^{24} + x$  en adelante,  $2^{23} + x$  se puede representar.

## Ejercicios - Respuesta

Demuestre que los números de punto flotante del tipo *float* del estándar IEEE754 **no** cumplen el principio de asociatividad de la suma, *i.e.*,  $x + (y + z) = (x + y) + z$

Esto se debe al error de redondeo de punto flotante. Usaremos una idea similar a la pregunta anterior.

Tomemos  $x = -2^{24}$ ,  $y = 2^{24}$ ,  $z = 1$

Veamos su representación de punto flotante en IEEE754.

## Ejercicios - Respuesta

$$x = -2^{24}$$

$$= (-1)^1 * 1.000000000000000000000000 * 10^{\text{exp}} \rightarrow \text{exp} - 127 = 24$$

$$= (-1)^1 * 1.000000000000000000000000 * 10^{10010111} \rightarrow \text{exp} = 151 = 10010111\text{b}$$

$$x_{\text{IEEE754}} = 11001011100000000000000000000000$$

$$y = 2^{24}$$

$$= (-1)^0 * 1.000000000000000000000000 * 10^{\text{exp}} \rightarrow \text{exp} - 127 = 24$$

$$= (-1)^0 * 1.000000000000000000000000 * 10^{10010111} \rightarrow \text{exp} = 151 = 10010111\text{b}$$

$$y_{\text{IEEE754}} = 01001011100000000000000000000000$$

## Ejercicios - Respuesta

$$z = 1$$

$$= (-1)^0 * 1.000000000000000000000000 * 10^{\text{exp}} \rightarrow \text{exp} - 127 = 0$$

$$= (-1)^0 * 1.000000000000000000000000 * 10^{01111111} \rightarrow \text{exp} = 127$$

$$z_{\text{IEEE754}} = 00111111000000000000000000000000$$

En el primer término, es trivial ver que como  $y = -x$ , entonces  $(x + y)$  será 0 y por ende:

$$(x + y) + z = z = 1$$

Ahora, ¿qué pasa con el segundo término? Veamos.



## Ejercicios - Respuesta

$$\begin{aligned}
 y + z &= (-1)^0 * 1.000000000000000000000000 * 10^{(24)b} \\
 &\quad + (-1)^0 * 1.000000000000000000000000 * 10^{(0)b} && \text{Igualamos exponentes.} \\
 &= (-1)^0 * 1.000000000000000000000000 * 10^{(24)b} \\
 &\quad + (-1)^0 * 0.000000000000000000000001 * 10^{(24)b} \\
 &= (-1)^0 * 1.000000000000000000000001 * 10^{(24)b} && \text{Aplicamos desfase de exponente.} \\
 &= (-1)^0 * 1.000000000000000000000001 * 10^{10010111} \\
 (y + z)_{\text{IEEE754}} &= 01001011100000000000000000000000
 \end{aligned}$$

Como solo podemos utilizar 23 bits para la mantisa, se pierde el bit menos significativo (equivalente al aporte de  $z$ ) y tenemos que  $y + z = y$ .

De esta forma:  $x + (y + z) = x + y = 0 \neq (x + y) + z = z = 1$ .