DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

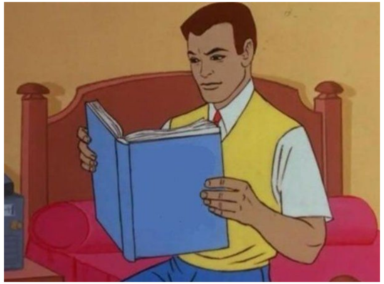
Clase 6 - Representaciones Numéricas II

Profesor: Germán Leandro Contreras Sagredo

Bibliografía

- **Apuntes históricos. Hans Löbel, Alejandro Echeverría**
  - 6.5 - Representaciones Numéricas Parte 2 - Números Racionales

- **D. Patterson, Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan Kaufmann, 2020.**
  - Capítulo 3.5. Página 191, 265 en PDF.



Objetivos de la clase

- Entender el paso de base decimal a binaria para números reales.
- Conocer distintas representaciones de números reales 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 reales?

- 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 reales

¿Cómo representamos los números reales de forma posicional?
Realizando la expansión con exponentes negativos.

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

Podemos hacer lo mismo en base binaria.

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

Ahora, ¿cómo pasamos de base decimal a binaria?

Representaciones de números reales - División binaria

Podemos obtener el valor en binario de un número real 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 reales - División binaria

Veremos a continuación la división binaria del ejemplo anterior.

11 : 100 = **0.**
11**(0)**

— — — — — — — —

110 : 100 = 0.**1**
-100
010(0)

— — — — — — — —

100 : 100 = 0.1**1**
000

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.

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.

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

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

Representaciones de números reales - División binaria

Ahora… ¿Cómo obtenemos el valor 0.1 en binario?

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

1 : 1010 = **0.000**
1**(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.

10000 : 1010 = 0.000**1**
-01010
00110(0)
— — — — — — — —

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

1100 : 1010 = 0.0001**1**
-1010
0010(0)

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 reales - División binaria

100 : 1010 = 0.00011**00**
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 = 0.0001100**1**
-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 = 0.00011001**1**
-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 reales - 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.

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

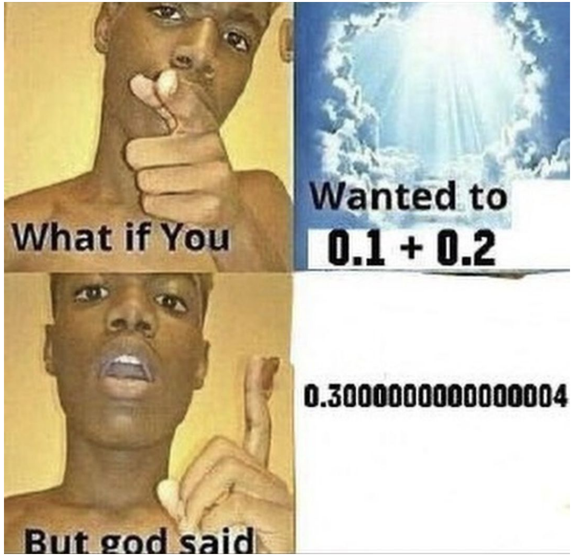
En resumen, la base escogida puede afectar en la finitud del número real que se busca representar.

Representaciones de números reales - 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.




```
>>> 0.3 - 0.1 == 0.2
False
```

Ejemplo en Python 3.9.

Representaciones de números reales - 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 reales

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

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

Representaciones de números reales - 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 **siempre** en la misma posición.

Ejemplo

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

10,110b = $0_s010_t1100_f$ -1,0011b = $1_s001_t0011_f$

Representaciones de números reales - 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 reales - 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

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

* En base binaria, siempre normalizamos dejando un bit igual a 1 a la izquierda del punto decimal.

¿Y en base binaria?

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

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 reales - 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*)

* Al normalizar, flotaremos el punto hacia el primer bit igual a uno

$10{,}110b = 1{,}01\mathbf{1} * 10^{001} = 0_{ss}101_{s}0_{se}001_{e}$

Perdemos el bit menos significativo por redondeo

$-0{,}00011b = -1{,}1 * 10^{-100} = 1_{ss}110_{s}1_{se}100_{e}$

Representaciones de números reales - 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.

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 significante parte en 1** (*i.e.* 1.*xxx*), por ende, se desprende del uso de dicho bit. También se le llama **mantisa**.

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 00000000000000000000000.

* El cero es el único número con ningún bit igual a 1.

- 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 = 011111111*xxxxxxxxxxxxxxxxxxxxxxx*

- En el caso de NaN (***N**ot **A** **N**umber*), 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 0xE 0x2 0x0 0x0 0x0 0x0 0x0

- Usando el algoritmo visto en la clase 1, podemos obtener ambos formatos rápidamente.

Representación de números de punto flotante - IEEE754

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

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

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

En este caso, *exp* - 127 = -3 → *exp* = 124 = 01111100b

Siguiendo el estándar: $0.00101b = (-1)^{0} * 1.01 * 10^{01111100}$

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

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.

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 seg. = 103100 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 - Más opciones

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 reales con gran cantidad de decimales.

- **Desventaja:** Tiempo de procesamiento aún mayor.

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

Representación de números de punto flotante - Operaciones

Supongamos que tenemos tres números reales binarios . ¿Cómo podemos operar con ellos?

Ejemplos: $A = 1{,}11 * 10^{11}$; $B = 1{,}1 * 10^{11}$; $C = 1{,}11 * 10^{1}$

- $A + B = 11{,}01 * 10^{11} = 1{,}101 * 10^{100}$
  - Al tener los mismos exponentes, solo sumamos los significantes y normalizamos.
- $A + C =$ **¿?** $A * B =$ **¿?** $A * C =$ **¿?**
  - ¿Qué hacemos con la suma si no se cumple el requisito antes descrito? ¿Cómo multiplicamos dos números reales en binario?

Representación de números de punto flotante - Operaciones

Suma: Suponga que desea sumar $A = 1{,}11 * 10^{11}$ y $C = 1{,}11 * 10^{1}$

1. Si sus exponentes son distintos, tome el número de menor exponente y realice lo siguiente:
   1.1. Haga un *shift right* sobre los bits de significante.
   1.2. Incremente en una unidad los bits de exponente.
   1.3. Repita este procedimiento hasta que los exponentes de ambos números sean iguales.
      - **Ej:** $C = 1{,}11 * 10^{1} =^{SHR} 0{,}111 * 10^{10}$
        $=^{SHR} 0{,}0111 * 10^{11}$

Representación de números de punto flotante - Operaciones

Suma: Suponga que desea sumar $A = 1{,}11 * 10^{11}$ y $C = 1{,}11 * 10^{1}$

2. Sume los significantes, manteniendo el exponente.
   - **Ej:** $(1{,}11 * 10^{11})_A + (0{,}0111 * 10^{11})_C = \quad 10{,}0011 * 10^{11}$
3. Normalice el resultado, ya sea mediante *shifts* a la izquierda (disminución del exponente) o a la derecha (aumento del exponente).
   - $A + C \quad = \quad 10{,}0011 * 10^{11}$

     $=^{SHR}$ $\mathbf{1{,}00011 * 10^{100}}$

Representación de números de punto flotante - Operaciones

Suma: Suponga que desea sumar $A = 1{,}11 * 10^{11}$ y $C = 1{,}11 * 10^{1}$

4. (Opcional) En caso de tener más bits de significante respecto a los que se pueden almacenar, **redondear**.
   - Distintas alternativas:
     - Truncado.
     - Redondeo hacia arriba o hacia abajo.
     - Redondeo hacia el **par** más cercano.
   - Trabajando con floats, se almacenan hasta 23 bits (24 si se considera el bit implícito anterior al punto decimal).

Representación de números de punto flotante - Operaciones

Suma: Suponga que desea sumar $A = 1{,}11 * 10^{11}$ y $C = 1{,}11 * 10^{1}$

Verifiquemos nuestro resultado:

$A = 1{,}11 * 10^{11} = 1110 = 14_{10}$; $C = 1{,}11 * 10^{1} = 11{,}1 = 3{,}5_{10}$

$\rightarrow \quad (A + C)_{10} = 17{,}5_{10}$

$$
\begin{aligned}
A + C &= 1{,}00011 * 10^{100} \\
&= 10001{,}1 \\
&= (1 * 2^{4} + 1 * 2^{0} + 1 * 2^{-1})_{10} = (16 + 1 + 0{,}5)_{10} = 17{,}5_{10}
\end{aligned}
$$

Representación de números de punto flotante - Operaciones

Multiplicación: Suponga que desea multiplicar $A = 1{,}11 * 10^{11}$ y $B = 1{,}1 * 10^{11}$

1. Sume los exponentes.
   1.1. Si están con desfase (por ejemplo, floats), restar dicho desfase en la suma.
      - **Ej:** $(exp_A + des)_A + (exp_B + des)_B - des = exp_A + exp_B + des$

   1.2. Si no están con desfase, sumar directo:
      - **Ej:** $exp_A + exp_B = 11 + 11 = 110$

* exp_i = Exponente de i
* des = Desfase

Representación de números de punto flotante - Operaciones

Multiplicación: Suponga que desea multiplicar $A = 1{,}11 * 10^{11}$ y $B = 1{,}1 * 10^{11}$

2. Multiplique los significantes.
   - Multiplique e ignore el punto decimal. Luego, posicione el punto sobre el resultado según el conteo de posiciones decimales del multiplicando y el multiplicador.
   - **Ej:**

		1 ,	1	1	x	1 ,	1
			2				1
		1	1	1			
+	1	1	1				
1	0 ,	1	0	1			
			2 + 1 = 3				

* El punto decimal se mueve tres posiciones decimales ya que el multiplicando posee dos posiciones decimales; mientras que el multiplicador posee una (1 + 1 = 2). Esto es equivalente a: $(111 * 10^{-2}) * (11 * 10^{-1}) = (111 * 11) * 10^{-3}$

Representación de números de punto flotante - Operaciones

Multiplicación: Suponga que desea multiplicar $A = 1{,}11 * 10^{11}$ y $B = 1{,}1 * 10^{11}$

3. Normalice el resultado, ya sea mediante *shifts* a la izquierda (disminución del exponente) o a la derecha (aumento del exponente). Recuerde realizar esto sobre el exponente obtenido en el paso 1.
   - $A * B \quad = \quad 10{,}101 * 10^{110}$

     $=^{SHR} \mathbf{1{,}0101 * 10^{111}}$

Representación de números de punto flotante - Operaciones

Multiplicación: Suponga que desea multiplicar $A = 1{,}11 * 10^{11}$ y $B = 1{,}1 * 10^{11}$

4. (Opcional) En caso de tener más bits de significante respecto a los que se pueden almacenar, **redondear**.
   - Se tienen las mismas opciones que con la suma.

Representación de números de punto flotante - Operaciones

Multiplicación: Suponga que desea multiplicar $A = 1{,}11 * 10^{11}$ y $B = 1{,}1 * 10^{11}$

Verifiquemos nuestro resultado:

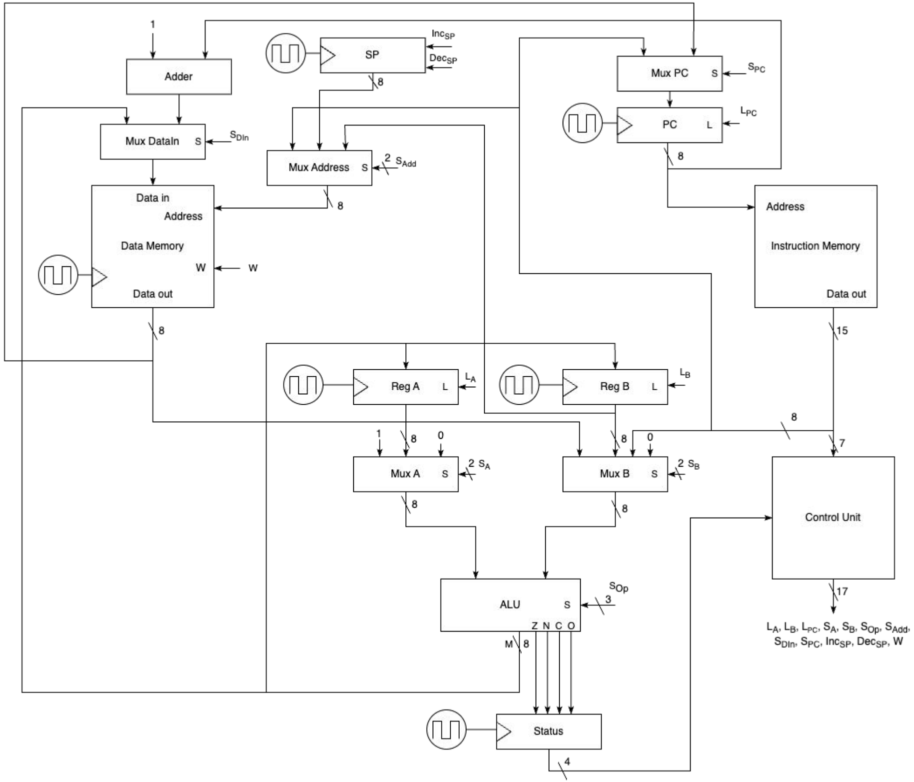
$A = 1{,}11 * 10^{11} = 1110 = 14_{10}$; $B = 1{,}1 * 10^{11} = 1100 = 12_{10}$
$\rightarrow \ (A * B)_{10} = 168_{10}$

$$
\begin{aligned}
A * B \quad &= 1{,}0101 * 10^{111} \\
&= 10101000 \\
&= (1 * 2^{7} + 1 * 2^{5} + 1 * 2^{3})_{10} = (128 + 32 + 8)_{10} = 168_{10}
\end{aligned}
$$

Computador básico - Punto flotante

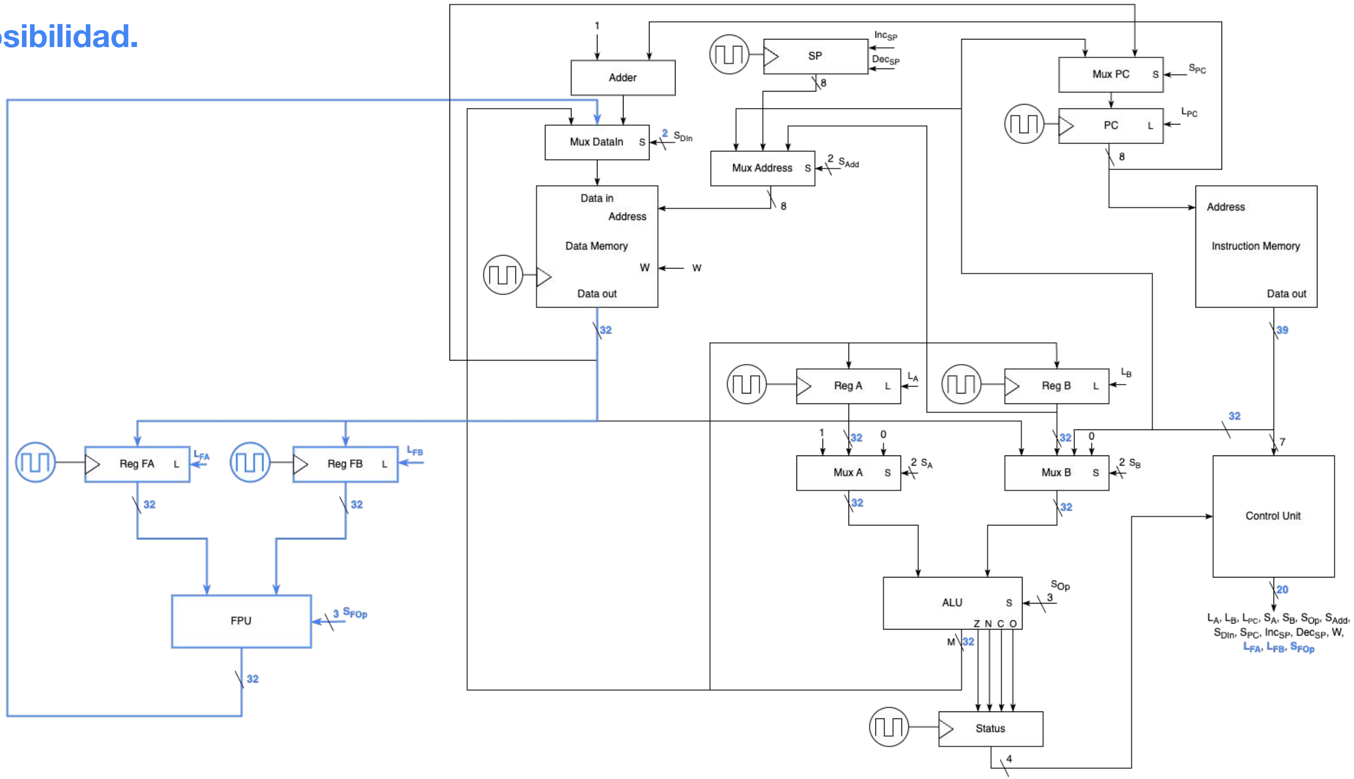
¿Cómo podríamos modificar el diagrama de nuestro computador para soportar números de punto flotante?

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



Computador básico - Punto flotante

Esta es solo una posibilidad.



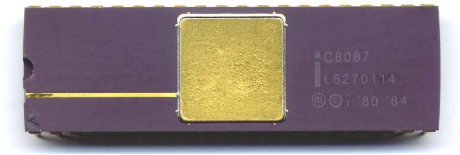
Computador básico - Punto flotante

- 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.

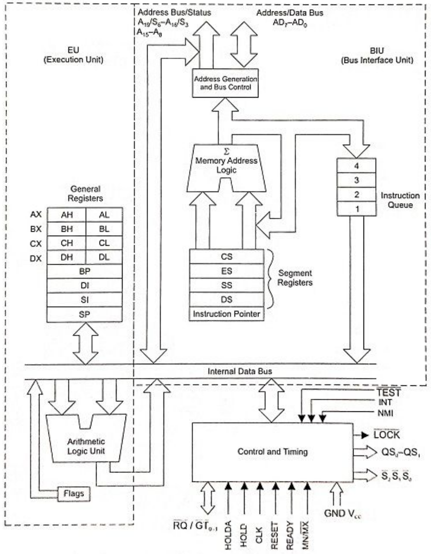
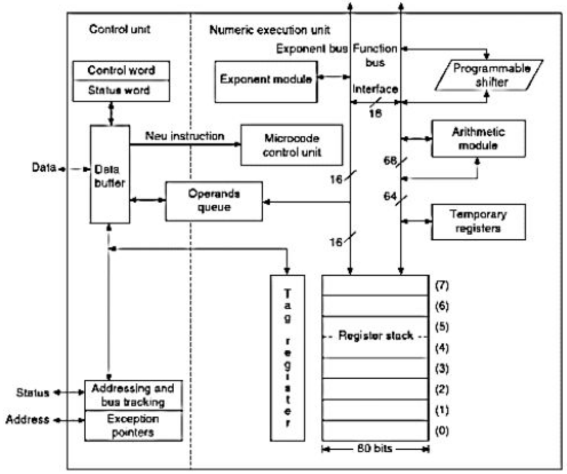


Fig. 5.24 *Block diagram of 8088 microprocessor*



Block Diagram of 8087 math coprocessor

Diagrama de bloques de ambos procesadores.

Errores comunes

A continuación, veremos los errores más comunes que cometen en las interrogaciones de forma que puedan evitarlos.

Errores comunes

Aplicar mal el desfase de exponente al guardar o leer un float

- Cuando se almacena un número real en formato float, su exponente **incrementa** en 127 unidades. Por ende, al leerlo de vuelta, se debe **decrementar** en 127 unidades para obtener su valor real.

- Un error común es realizar estas operaciones **al revés**.

Errores comunes

Aplicar mal el desfase de exponente al guardar o leer un float

- **Ejemplo:** Almacenar en formato float el número 1,01 * $10^{01111111}$

$exp = 127$
$exp_{float} = 127 - 127 = 00000000$

$exp = 127$
$exp_{float} = 127 + 127 = 11111110$

INCORRECTO

Al leer el número de vuelta y aplicar el desfase, tendríamos un exponente igual a -127, lo que es incorrecto.

CORRECTO

Al leer el número de vuelta y aplicar el desfase, tendremos un exponente igual a 127, que es el valor original.

Errores comunes

Aplicar mal el desfase de exponente al guardar o leer un float

- **Ejemplo:** Leer de vuelta el float de 1,01 * $10^{01111111}$
  Su representación es: 0 11111110 01000000000000000000000

$\texttt{exp}_{\texttt{float}}$ = 254
$\texttt{exp}$ = 254 + 127 = 381

$\texttt{exp}_{\texttt{float}}$ = 254
$\texttt{exp}$ = 254 - 127 = 01111111

INCORRECTO
En este caso, resulta en un número que ni siquiera se puede representar con 8 bits.

CORRECTO
Se obtiene el valor almacenado originalmente.

Errores comunes

No aplicar desfase al leer un float

- Otro error común es **no aplicar el desfase al interpretar el exponente del float**.

- **Ejemplo:** Interprete el exponente del siguiente float:

  0**11111110**01000000000000000000000

exp = 254	exp = 254 - 127 = 01111111 = 127

INCORRECTO
No se consideró el desfase.

CORRECTO

Errores comunes

Indicar pérdida de precisión de forma errónea

- Si se le pide cuantificar la pérdida de precisión en una representación, es importante **contar bien los bits perdidos**.

- **Ejemplo:** Indique el error de precisión en una representación de punto fijo con 3 bits fraccionales para el número 101,00101

$\underbrace{101{,}001\mathbf{01}}_{\text{Potencias } 2^{-4} \text{ y } 2^{-5}}$ → Se pierden los dos bits menos significativos

Pérdida: 0,00001 = 2^{-5} = 0,03125

Es **muy común** contar mal los bits que se pierden.

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 $A : B$?

***Interrogación* 1, 2011-2**

Ejercicios

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

***Examen*, 2016-1**

Ejercicios

Indique si los siguientes números pueden representarse o no como un número *float* bajo el estándar IEEE754 con su valor exacto:

I. $2^{26} + 2$
II. 2^{70}

Si se puede representar, indique la representación; en otro caso, justifique por qué no se puede.

***Examen*, 2023-1**

Ejercicios

Suponga que se crea el tipo de dato `arquiFloat` de 32 bits a partir del nuevo estándar "IIC2343" para representar números de punto flotante. A diferencia del tipo de dato `float` del estándar IEEE754, este posee el siguiente formato:

- Un bit de signo de significante.
- Un bit de signo de exponente.
- 10 bits de exponente **no desfasado**.
- 20 bits de significante **normalizado** con bit implícito.

Compare los dos estándares respondiendo las siguientes preguntas:

1. Señale una ventaja y una desventaja del tipo de dato `arquiFloat` respecto del tipo de dato `float`.
2. Señale cómo se representa el resultado de la operación $2^{22} + 7$ en formato `float` y `arquiFloat`. ¿Existe pérdida de precisión en alguno de los casos?

Interrogación 1, 2023-2

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.*, $x + (y + z) = (x + y) + z$

Interrogación **1, 2014-2**

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 = 00111110001000000000000000000000

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:** 00000000000000000000000.

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

Indique si los siguientes números pueden representarse o no como un número *float* bajo el estándar IEEE754 con su valor exacto:

I. $2^{26} + 2$
II. 2^{70}

Si se puede representar, indique la representación; en otro caso, justifique por qué no se puede.

Respuesta en la siguiente diapositiva.

Ejercicios - Respuesta

Solución: Debemos considerar que el tipo de dato `float` del estándar IEEE754 puede tener una precisión exacta de hasta 23 decimales por su mantisa.

(i) El valor $2^{26} + 2$ requiere un bit igual a 1 en la posición 25 del significante ya que $2^{26} = 100000000000000000000000000b$ y $2 = 10b$. Por este motivo, su representación como `float` pierde los 3 bits menos significativos y terminaría por almacenar el valor 2^{26}.

(ii) Como este número no requiere bits de significante, se puede representar solo con su exponente. Se toma el exponente desfasado en 127, que es igual a $197 = 11000101b$. Así, $(2^{70})_{\text{IEEE754}} = 01100010100000000000000000000000$.

Ejercicios

Suponga que se crea el tipo de dato `arquiFloat` de 32 bits a partir del nuevo estándar "IIC2343" para representar números de punto flotante. A diferencia del tipo de dato `float` del estándar IEEE754, este posee el siguiente formato:

- Un bit de signo de significante.
- Un bit de signo de exponente.
- 10 bits de exponente **no desfasado**.
- 20 bits de significante **normalizado** con bit implícito.

Compare los dos estándares respondiendo las siguientes preguntas:

1. Señale una ventaja y una desventaja del tipo de dato `arquiFloat` respecto del tipo de dato `float`.
2. Señale cómo se representa el resultado de la operación $2^{22} + 7$ en formato `float` y `arquiFloat`. ¿Existe pérdida de precisión en alguno de los casos?

Respuesta en la siguiente diapositiva.

Ejercicios

1. Señale una ventaja y una desventaja del tipo de dato `arquiFloat` respecto del tipo de dato `float`.

Una ventaja es que `arquiFloat` posee un rango mayor respecto a los números que puede representar dada su cantidad de bits de exponente. En este caso, el mayor exponente representable es 1111111110b = 1022 (si se asume la reserva para los valores infinitos y con bit de signo positivo), que es significativamente mayor al exponente máximo de `float` igual a 127. Por otra parte, una desventaja es la pérdida de precisión. En este caso, si incluimos el bit implícito, `arquiFloat` puede representar números de hasta 21 bits en comparación a `float` que puede representar números de hasta 24 bits.

Ejercicios

2. Señale cómo se representa el resultado de la operación $2^{22} + 7$ en formato `float` y `arquiFloat`. ¿Existe pérdida de precisión en alguno de los casos?

El número que se busca representar, haciendo la suma entre 2^{22} y 7 con notación científica, es igual a $1.0000000000000000000111 * 10^{00010110}$. A continuación, veremos cómo se almacena en cada representación:

- **`float`:** El número es positivo (bit de signo igual a cero) y el exponente desfasado es igual a 22 + 127 = 149. Por lo tanto, se almacena la secuencia: $0_{\text{signo}}10010101_{\text{exp.}}00000000000000000001110_{\text{sig.}}$. No se pierde precisión.
- **`arquiFloat`:** Tanto el número como su exponente son positivos (bits de signo iguales a cero). Por lo tanto, se almacena la secuencia: $0_{\text{signo sig.}}0_{\text{signo exp.}}0000010110_{\text{exp.}}00000000000000000001_{\text{sig.}}$. En este caso, dado que no se explicita, no importa el orden en el que se guarden las secuencias de bits. Se pierden dos bits de precisión, por lo que se pierde el valor exacto del número: se redondea a $2^{22} + 4$.

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

$x = -2^{24}$

$= (-1)^{1} * 1.00000000000000000000000 * 10^{exp} \rightarrow exp - 127 = 24$

$= (-1)^{1} * 1.00000000000000000000000 * 10^{10010111} \rightarrow exp = 151 = 10010111b$

$x_{IEEE754} = 11001011100000000000000000000000$

$y = 2^{24}$

$= (-1)^{0} * 1.00000000000000000000000 * 10^{exp} \rightarrow exp - 127 = 24$

$= (-1)^{0} * 1.00000000000000000000000 * 10^{10010111} \rightarrow exp = 151 = 10010111b$

$y_{IEEE754} = 01001011100000000000000000000000$

Ejercicios - Respuesta

$z = 1$

$= (-1)^{0} * 1.00000000000000000000000 * 10^{exp} \rightarrow exp - 127 = 0$

$= (-1)^{0} * 1.00000000000000000000000 * 10^{01111111} \rightarrow exp = 127$

$z_{IEEE754} = 00111111100000000000000000000000$

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

$$
\begin{aligned}
&y + z \\
&\quad = (-1)^0 * 1.000000000000000000000000 * 10^{(24)b} \\
&\quad\quad + (-1)^0 * 1.000000000000000000000000 * 10^{(0)b} \\
&\quad = (-1)^0 * 1.000000000000000000000000 * 10^{(24)b} \\
&\quad\quad + (-1)^0 * 0.000000000000000000000001 * 10^{(24)b} \\
&\quad = (-1)^0 * 1.000000000000000000000001 * 10^{(24)b} \\
&\quad = (-1)^0 * 1.000000000000000000000001 * 10^{10010111} \\
&(y + z)_{IEEE754} = 01001011100000000000000000000000
\end{aligned}
$$

Igualamos exponentes.

Aplicamos desfase de exponente.

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

Anexo - Medición de pérdida de precisión de 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**.

Anexo - Medición de pérdida de precisión de punto flotante

Machine epsilon o *epsilon* de la máquina

Bajo esta representación:

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

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

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

Anexo - Medición de pérdida de precisión de 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:

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

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

Anexo - Medición de pérdida de precisión de punto flotante

Machine epsilon o *epsilon* de la máquina

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

$x = 1111b$

$y = 0_{ss}111_{s}0_{se}010_{e} = 111b$

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

Anexo - Medición de pérdida de precisión de punto flotante

Float y Double

- En el caso de un **float**, ε_{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:

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

- En el caso de un **double**, en cambio, ε_{mach} es igual a 10^{-52}.

Anexo - Números subnormales en punto flotante

Con el fin de poder representar números más pequeños y cercanos a cero bajo el estándar IEEE754, se definen los **números subnormales**. Para ello, se modifican las reglas de la siguiente forma:

- Si el exponente es igual a 0, **el bit implícito será igual a cero**.
- Si el exponente es igual a cero, su valor **quedará fijo en -126 y no en -127**.

Anexo - Números subnormales en punto flotante

De esta forma:

- La combinación de exponente = 0 y significante = 0 bajo el estándar **sigue siendo igual a cero (`0.0`)**.

- El cero se convierte en un **número subnormal**.

- El número más pequeño que no es subnormal es igual a:
  00000000100000000000000000000000
  $= (-1)^{0} * 1.00000000000000000000000 * 10^{-126}$
  $= 1 * 2^{-126}$

Anexo - Números subnormales en punto flotante

Por otra parte:

- El número subnormal más grande corresponde a:
  00000000011111111111111111111111
  $= (-1)^0 * 0.11111111111111111111111 * 10^{-126}$
  $= 0.99999988079071 * 2^{-126}$ → **Muy cercano al "normal" más pequeño**

- El número subnormal más pequeño corresponde a:
  00000000000000000000000000000001
  $= (-1)^0 * 0.00000000000000000000001 * 10^{-126}$
  $= 0.000002 * 2^{-126}$ → **Muy cercano al cero**

Anexo - Números subnormales en punto flotante

¿Por qué se usan entonces?

- Al momento de hacer cálculos, su uso permite que la disminución hacia el valor 0 sea más gradual y no abrupta.
- Reduce la cantidad de errores causada por la división por cero al poder representar divisores más pequeños.
- Si bien sacrifican precisión al no usar un bit implícito igual a 1, se acepta en pos de los beneficios que traen en términos de cálculos numéricos.

Anexo - Videos complementarios

Por si desean profundizar más en los temas de esta clase.

- Por qué los computadores se equivocan en la aritmética con *floats*.
- Profundización sobre el caso del misil Patriot.
- Profundización en números subnormales.
- Mario 64: *Crashing* del juego por redondeo de *floats*.