



Módulo 06

Norma IEEE-754

(Pt. 2)



Organización de Computadoras
Depto. Cs. e Ing. de la Comp.
Universidad Nacional del Sur



Copyright

- Copyright © **2011-2024** A. G. Stankevicius
- Se asegura la libertad para copiar, distribuir y modificar este documento de acuerdo a los términos de la **GNU Free Documentation License**, Versión 1.2 o cualquiera posterior publicada por la Free Software Foundation, sin secciones invariantes ni textos de cubierta delantera o trasera
- Una copia de esta licencia está siempre disponible en la página <http://www.gnu.org/copyleft/fdl.html>
- La versión transparente de este documento puede ser obtenida de la siguiente dirección:

<http://cs.uns.edu.ar/~ags/teaching>



Contenidos

- Teoría de redondeo
- Redondeos óptimos y simétricos
- Norma **IEEE-754**
- Formatos contemplados
- Concepto de denormal
- Implementación de las operaciones básicas
- Redondeos en la norma
- Consideraciones al implementar en **HW**

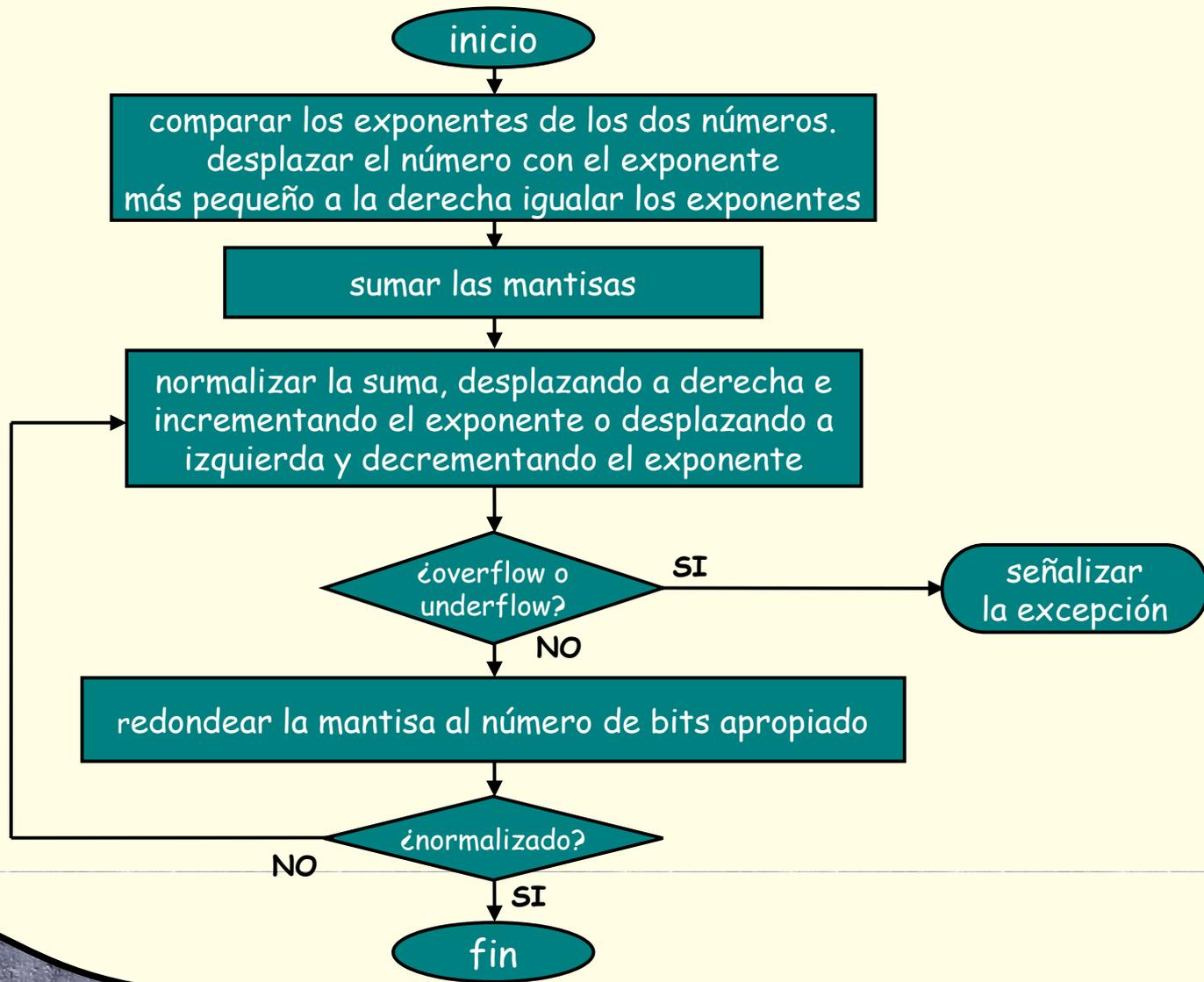


Suma en la norma

- Algoritmo básico para la suma de dos números en la norma **IEEE-754**:
 - ➔ Alinear los puntos decimales (“desnormalizando” a uno de los operandos de forma tal que ambos exponentes sean iguales), para esto se desplaza a la derecha la mantisa con el exponente más pequeño
 - ➔ Sumar las mantisas
 - ➔ Normalizar el resultado, verificando el eventual overflow en el exponente
 - ➔ Redondear el resultado al número correcto de dígitos significativos



Suma en la norma

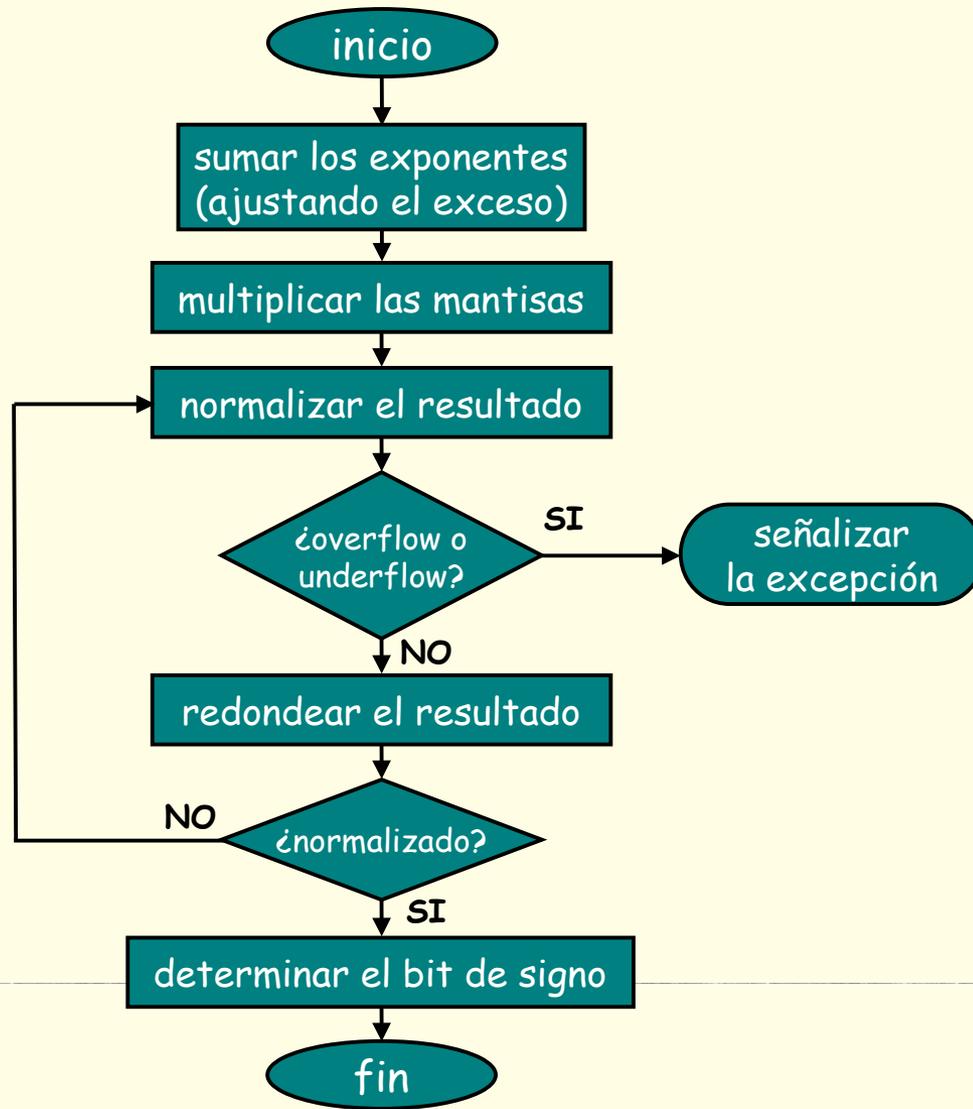


Multiplicación en la norma

- Algoritmo básico para la multiplicación de dos números en la norma **IEEE-754**:
 - ➔ Sumar los exponentes, compensando el exceso
 - ➔ Multiplicar las mantisas
 - ➔ Normalizar el resultado y comprobar si se produjo overflow
 - ➔ Redondear la respuesta al número correcto de dígitos significativos
 - ➔ Normalizar nuevamente en caso de ser necesario
 - ➔ Determinar el signo del resultado



Multiplicación en la norma



Redondeos en la norma

- La norma **IEEE-754** contempla **cinco modalidades de redondeo**, a saber:
 - Hacia arriba (hacia $+\infty$)
 - Hacia abajo (hacia $-\infty$)
 - Hacia el cero (truncado)
 - Proximidad unbiased (hacia los pares)
 - Proximidad biased (hacia afuera del cero)
- Estas modalidades de redondeo requieren el uso de **bits adicionales de precisión**



Bits de guarda

- Denominaremos **bits de guarda** a los bits de precisión adicionales a la derecha del bit menos significativo:
 - Los bits de guarda son conservados temporalmente hasta que sean requeridos
 - Usualmente son empleados en la normalización y resultan fundamentales durante el redondeo
 - La norma **IEEE-754** contempla tres bits de guarda, denominados guard (**G**), round (**R**), y sticky (**S**)
 - Nótese que esto implica que la **ALU** debe ensancharse en tres bits



Bits de guarda

- La norma **IEEE-754** al hacer un uso acertado de los bits de guarda logra obtener **el mismo resultado que se hubiera obtenido al hacer uso de una precisión infinita**
- En otras palabras, con sólo tres bits de guarda se puede computar el mismo resultado que se obtendría al operar sobre la totalidad de los bits de los operandos
- ➔ Hasta para la precisión óctuple que cuenta con **237** bits de mantisa apenas necesitás **3** bits adicionales



Round y sticky

- Por caso, analicemos cómo a través de sólo dos bits de guarda se puede obtener el redondeo correcto en la siguiente suma con una precisión de dos dígitos decimales:

$$\begin{aligned}x_1 + x_2 &= 2.561 \times 10^0 + 2.34 \times 10^2 \\ &= 0.02561 \times 10^2 + 2.34 \times 10^2 \\ &= 2.36 \underline{561} \times 10^2\end{aligned}$$


R = 5 S ≠ 0



Round y sticky

- Contando con estos dígitos de guarda, es posible implementar un correcto redondeo:
 - Para $0 \leq R < 5$ se redondea para abajo
 - Para $5 < R \leq 9$ se redondea para arriba
 - Para $R = 5$, se mira el estado de **S**
- Con respecto al sticky, sólo importa si es cero o no (independientemente de cuál sea la base)
 - En la norma **IEEE-754**, **S** se calcula haciendo **OR** entre todos los bits descartados (esto es, los que no forman parte del resultado ni tampoco de **G** ni **R**)



Round y sticky

- La siguiente tabla resume el rol de los bits **R** y **S** en los distintos esquemas de redondeo (donde p_0 es el valor del dígito menos significativo):

redondeo	resultado ≥ 0	resultado < 0
hacia $-\infty$		+1 LSB si (R OR S)
hacia $+\infty$	+1 LSB si (R OR S)	
hacia el 0		
proximidad hacia pares	+1 LSB si ((R AND p_0) OR (R AND S))	+1 LSB si ((R AND p_0) OR (R AND S))
proximidad afuera del 0	+1 LSB si R	+1 LSB si R



Round y sticky

- A manera de ejemplo, veamos como se aplican estos esquemas a la hora de redondear a un dígito de precisión los siguientes números:

redondeo	+1.400	-1.600	+1.500	-2.500	+2.502
hacia $-\infty$	+1	-2	+1	-3	+2
hacia $+\infty$	+2	-1	+2	-2	+3
hacia el 0	+1	-1	+1	-2	+2
prox. a pares	+1	-2	+2	-2	+3
prox. fuera 0	+1	-2	+2	-3	+3



Round y sticky

- Finalmente, veamos en concreto cómo con sólo los bits **R** y **S** se obtiene el mismo resultado al redondear que al usar precisión infinita

→ Cálculo con precisión infinita:

$$x_1 = 1.001000 \times 2^3$$

$$x_2 = \underline{+1.101001} \times 2^{-1}$$



$$1.001000 \quad \times 2^3$$

$$+0.0001101001 \times 2^3$$

$$\underline{1.0011101001} \times 2^3$$

$$1.001111 \times 2^3$$



Round y sticky

Continúa:

→ Cálculo usando sólo round y sticky:

$$x_1 = 1.001000 \times 2^3$$

$$x_2 = \underline{+1.101001} \times 2^{-1}$$



$$1.001000 \times 2^3$$

$$\underline{+0.00011011} \times 2^3$$

$$1.00111011 \times 2^3$$

$$\left\{ \begin{array}{l} R = 1 \\ S = 0 \text{ OR } 0 \text{ OR } 1 = 1 \end{array} \right.$$

$$1.001111 \times 2^3$$



Rol del bit guard

- Si **R** y **S** se usan para redondear, ¿para qué se usa el tercer bit de guarda, el bit **G**?
 - El bit **G** se emplea al normalizar el resultado preliminar de las operaciones aritméticas, esto es, antes de aplicar el redondeo
 - En los casos en que no haga falta usar el bit **G**, se deben ajustar los valores de los bits **R** y **S** de la siguiente manera:

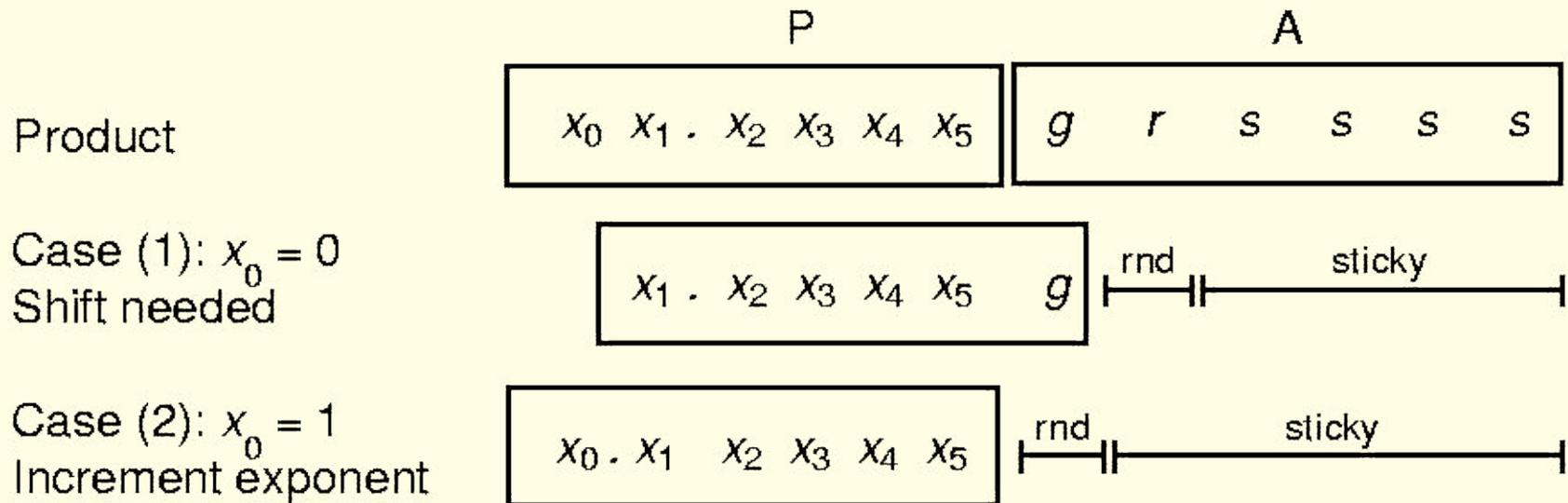
$$R_{\text{nuevo}} = G_{\text{anterior}}$$

$$S_{\text{nuevo}} = R_{\text{anterior}} \text{ OR } S_{\text{anterior}}$$



Rol del bit guard

- En la multiplicación, el rol del bit **G** se torna aparente cuando el bit más significativo del resultado preliminar es cero:



Adjust binary point,
add 1 to exponent to compensate



Implementación en HW

- Hasta ahora hemos repasado los algoritmos de las operaciones básicas haciendo hincapié en familiarizarnos con los mismos
- Naturalmente a la hora de implementar estos algoritmos en hardware surgen otras consideraciones:
 - ➔ **Alcanzar un desempeño altamente eficiente** (no tomar decisiones que comprometan el desempeño)
 - ➔ **Minimizar la cantidad de hardware necesario** (reusar en la medida de lo posible el hardware disponible)



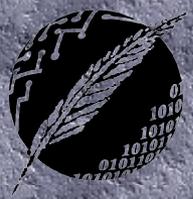
Implementación en HW

- Concretamente, la implementación en hardware de estos algoritmos capitalizará los beneficios de las distintas representaciones ya vistas:
 - ➔ Por caso, no tiene sentido contar con un circuito para sumar y otro independiente para la resta cuando se puede sacar provecho de las representaciones complementarias
 - ➔ Nótese que si bien la norma establece que la mantisa se representa en **SM**, **inada impide a la ALU que cambie internamente la representación de los operandos!**



Algoritmo para la suma

- A continuación repasaremos un algoritmo para la suma en punto flotante el cual incorpora la mejora antes comentada
- Sea $x_1 = (e_1, m_1)$ y $x_2 = (e_2, m_2)$ los dos operandos que se desean sumar:
 - Comparar los exponentes. En caso de que $e_1 < e_2$ intercambiar los operandos, a fin de que la diferencia $d = e_1 - e_2$ no sea negativa (será cero o positiva)
 - Si los signos de x_1 y x_2 difieren, reemplazar la mantisa m_2 por su complemento a dos



Algoritmo para la suma

Continúa:

- Colocar m_2 en un registro de desplazamiento y correr d lugares a la derecha. De los bits que se descartan por el extremo derecho armar los bits **G**, **R** y **S**
- Computar el resultado preliminar $S = m_1 + m_2$. Si los signos de los operandos son distintos y no hay carry de salida, reemplazar a **S** por su complemento a dos
- Normalizar a **S** de acuerdo al siguiente detalle: si los signos de los operandos son iguales y hay carry de salida, desplazar a **S** un lugar a la derecha, ingresando el **1** del carry (hubo overflow virtual)



Algoritmo para la suma

Continúa:

- Caso contrario, desplazar **S** a la izquierda hasta obtener una mantisa normalizada. Nótese que el primer desplazamiento hace ingresar a **G**, pero los restantes deben ingresar ceros
- En todos los casos, ajustar el exponente en función de la cantidad de desplazamientos realizados
- Ajustar **R** y **S** en caso de ser necesario, es decir, si **G** no pasó a formar parte de la mantisa en el último paso, hacer **$R = G$** y **$S = R \text{ OR } S$**



Algoritmo para la suma

● Continúa:

- ➔ Aplicar el redondeo elegido en función de los valores de los bits **R** y **S**. En caso de producirse carry out, desplazar la mantisa del resultado a derecha, ajustando el exponente de manera acorde
- ➔ Computar el signo del resultado. Si ambos operandos tienen el mismo signo, ese será el signo del resultado
- ➔ En caso contrario, se debe tener en cuenta cuál de los dos operandos era negativo, si se intercambiaron los operandos en el primer paso y de si se dos complementó el resultado preliminar



Algoritmo para la suma

- Los distintos parámetros que deben ser tenidos en cuenta a la hora de determinar el signo del resultado pueden resumirse en la siguiente tabla:

¿hubo swap?	¿se comp.?	signo(x_1)	signo(x_2)	signo(S)
SI		+	-	-
SI		-	+	+
NO	NO	+	-	+
NO	NO	-	+	-
NO	SI	+	-	-
NO	SI	-	+	+

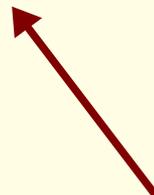


Ejemplo

- Calcular la siguiente suma, haciendo uso del redondeo por proximidad unbiased

$$\begin{aligned}
 x_1 &= +1.111111111111111111111111111111 \times 2^{45} \\
 x_2 &= +1.101001000000000000000000000000 \times 2^{21}
 \end{aligned}
 \left. \vphantom{\begin{aligned} x_1 \\ x_2 \end{aligned}} \right\} d = 24$$

$$\begin{array}{r}
 1.111111111111111111111111111111 \quad \times 2^{45} \\
 +0.000000000000000000000000000000111 \quad \times 2^{45} \\
 \hline
 1.111111111111111111111111111111111 \quad \times 2^{45} \\
 \text{GRS}
 \end{array}$$



el resultado preliminar está normalizado,
G no será usado en esta ocasión



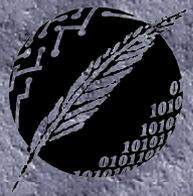
Ejemplo

- Calcular la siguiente suma, haciendo uso del redondeo por proximidad unbiased

$$\left. \begin{aligned} x_1 &= +1.01000000110100110100000 \times 2^{16} \\ x_2 &= -1.11001001101000011001000 \times 2^{16} \end{aligned} \right\} d = 0$$

$$\begin{array}{r} 1.01000000110100110100000 \quad \times 2^{16} \\ +0.00110110010111100111000000 \times 2^{16} \\ \hline 1.01110111001100011011000000 \times 2^{16} \\ \text{GRS} \end{array}$$

el resultado preliminar es negativo (no hubo carry), se debe complementar y se hará uso del bit **G**



Ejemplo

Continúa:

luego de complementar, se normaliza

$$0.10001000110011100101000000 \times 2^{16}$$

al normalizar, el bit **G** ingresa a la mantisa en el primer corrimiento

$$1.000100011001110010100000 \times 2^{15}$$

$R = 0$ y $S = 0$
no se suma nada

$$x_1 + x_2 = -1.000100011001110010100000 \times 2^{15}$$



para determinar el signo del resultado se tiene en cuenta que no hubo swap y que se complementó



Ejemplo

- Calcular el siguiente producto trabajando con sólo 9 bits de precisión, haciendo uso del redondeo hacia $-\infty$

$$x_1 = +1.01000000 \times 2^{-5}$$

$$x_2 = -1.10001001 \times 2^{32}$$

ajustar con cuidado
la posición del punto binario
del resultado preliminar

$$\begin{array}{r}
 1.01000000 \\
 \times 1.10001001 \\
 \hline
 101000000 \\
 101000000 \\
 101000000 \\
 + 101000000 \\
 \hline
 01.1110101101000000
 \end{array}$$

determinar **G**, **R** y **S** a partir
de los bits descartados

01.1110101101

GRS



Ejemplo

Continúa:

falta un bit, se debe hacer uso de **G**

R = 0 y **S = 1**
sumo **1 LSB**
(pues es < 0)

$$01.1110101101 \times 2^{27}$$

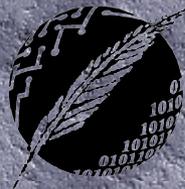
al normalizar, el bit **G** pasa a formar parte de la mantisa

$$1.11101011 \times 2^{27}$$

los exponentes se suman

el exponente no cambia al ingresar **G**

$$x_1 \times x_2 = -1.11101100 \times 2^{27}$$



Ejemplo

- Calcular el siguiente producto trabajando con sólo 9 bits de precisión, haciendo uso del redondeo hacia el 0

$$x_1 = +1.11000011 \times 2^{86}$$

$$x_2 = +1.10100001 \times 2^{12}$$

ajustar con cuidado
la posición del punto binario
del resultado preliminar

$$\begin{array}{r}
 1.11000011 \\
 \times 1.10100001 \\
 \hline
 111000011 \\
 111000011 \\
 111000011 \\
 + 111000011 \\
 \hline
 10.1101111010100011
 \end{array}$$

determinar **G**, **R** y **S** a partir
de los bits descartados

10.1101110011

GRS



Ejemplo

Continúa:

no normalizado,
no se hará uso de **G**
(ajustar **R** y **S**)

$$10.1101110011 \times 2^{98}$$

los exponentes
se suman

no se suma nada
(pues truncamos)

$$1.01101110 \times 2^{99}$$

correr la coma
a izquierda y
ajustar el exponente

$$x_1 \times x_2 = +1.01101110 \times 2^{99}$$



¿Preguntas?

