Funciones elementales

Procesado Digital de la Señal en FPGA

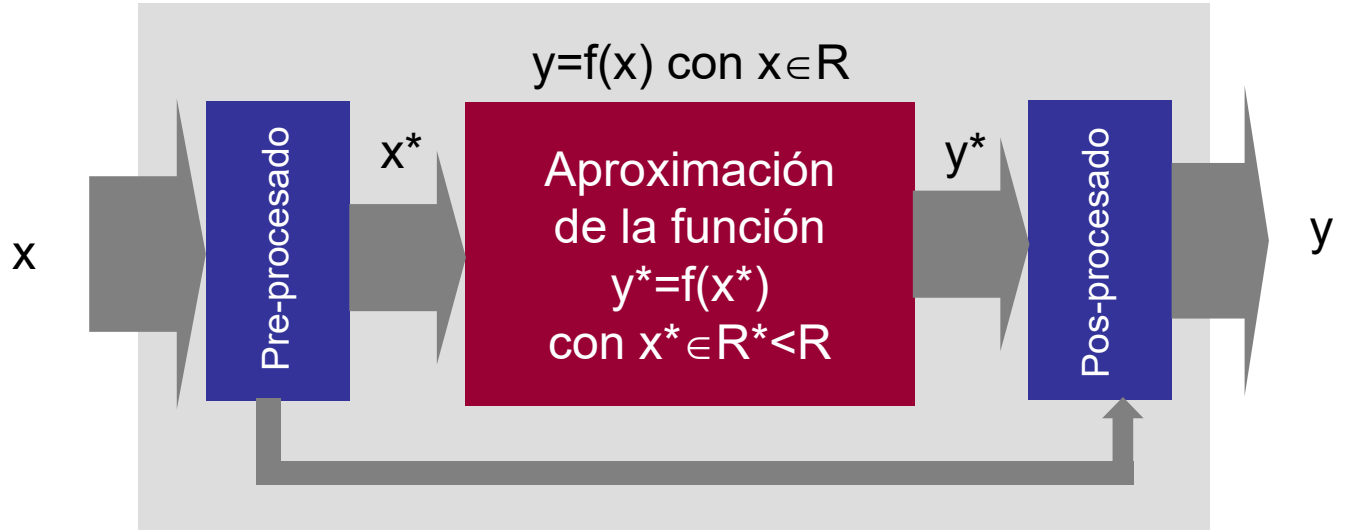
2021/2022

Funciones elementales

1. Aproximación de funciones elementales
2. Métodos basados en tablas
    1. Directo
    2. Interpolación
3. Métodos iterativos
    1. Newton-Raphson
    2. CORDIC
4. Evaluación polinómica
5. Reducción-ampliación del rango

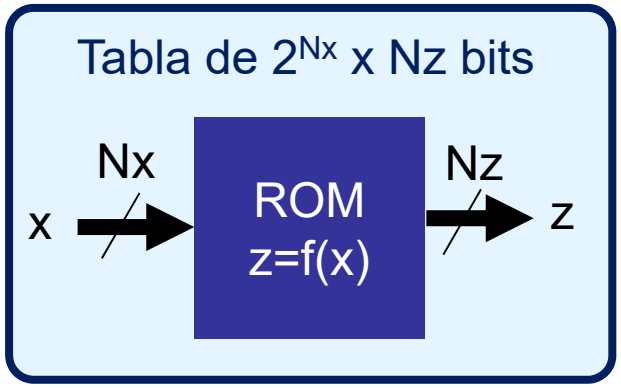
Aproximación de funciones

- Muchas aplicaciones requieren el cómputo de funciones elementales: 1/x, sqrt, log, exp, cos, sin, atan …
- No suele ser eficiente aproximar la función en todo el rango de aplicación (R)
- Algunos métodos solo convergen para entradas en un rango reducido (R*)
- Método general de aproximación de funciones:
  - Preprocesado: reducción del rango
  - Aproximación de la función en el rango reducido
  - Pos-procesado: reconstrucción del resultado, expansión al rango completo



Métodos basados en tablas

- **Memorias → implementación VLSI eficiente**
- **Método directo**: z=f(x)
  - x codificada con Nx bits
  - z codificada con Nz bits



Memorias en FPGAs:
- Basadas en LUT: memorias pequeñas de 16 o 64 bits
- Hard-cores: memoria grandes de 8k a 36kbits
  BRAMs en Xilinx y Mxk en Altera

Configuraciones M9K de Cyclone IV:
- 8Kx1 bits
- 4Kx2 bits
- 2Kx4 bits
- 1Kx(8+1) bits
- 512x(16+2) bits
- 256x(32+4) bits

Métodos basados en tablas

- **Método directo**: **Ej.1**: z=sin(2πx)
  - x con 11 bits
  - z con 12 bits

Tabla de 2^{11} x 12 bits

M9K Cyclone: 9kbits
⇒ 3 M9K configuración: 3x(2kx4) bits
BRAM Virtex 6: 36kbits
⇒ 1 BRAM configuración: 2kx16 bits

Ej.2: z=atan(y/x)
  - x,y con 11 bits
  - z con 12 bits

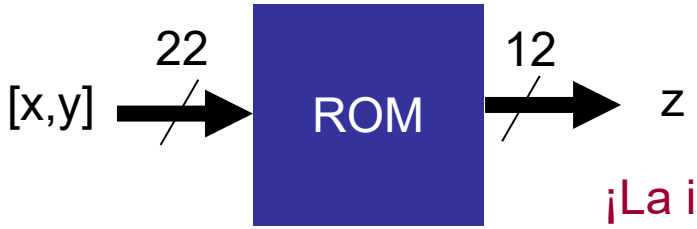


Tabla de 2^{22} x 12 bits

⇒ 6144 M9K Cyclone IV
⇒ 1536 BRAMs Virtex 6

¡La implementación directa no es un método razonable para tamaños de tablas grandes!

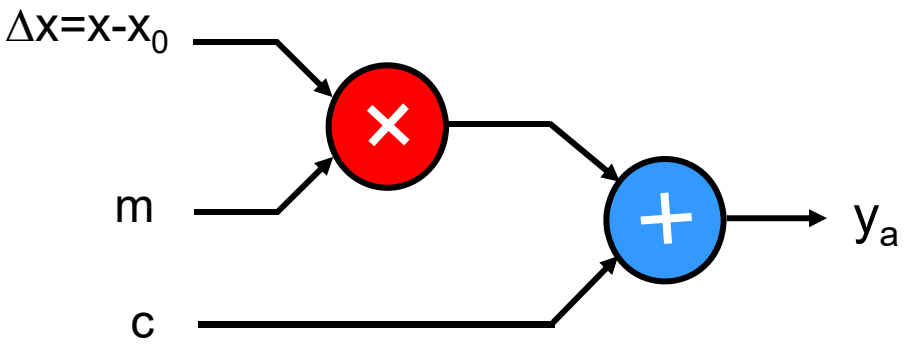
Métodos basados en tablas

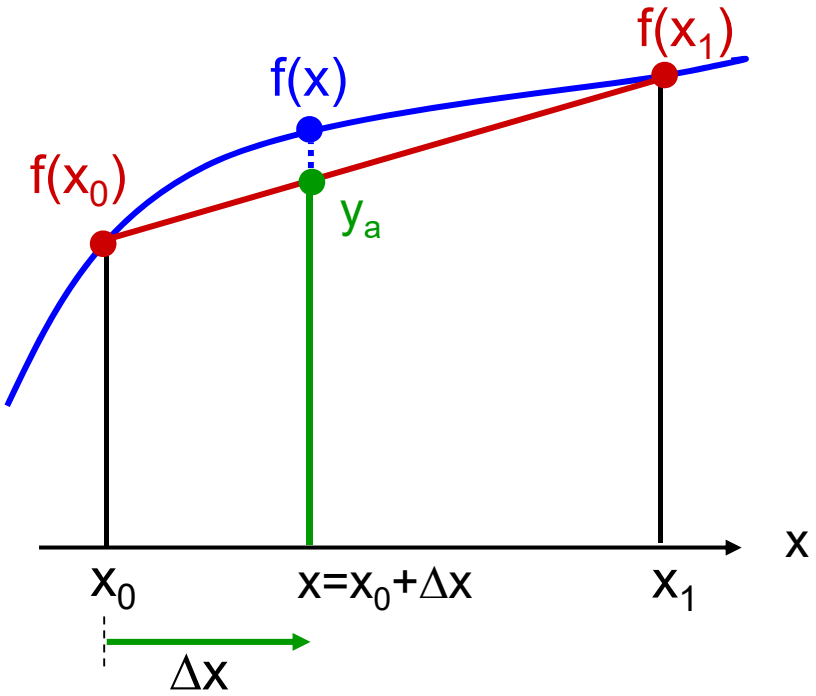
- **Interpolación lineal**: y=f(x)

  Aproximación en $[x_0,x_1] \Rightarrow y_a=m\cdot\Delta x+c$

  $m=(f(x_1)-f(x_0))/(x_1-x_0)$

  $c=f(x_0)$



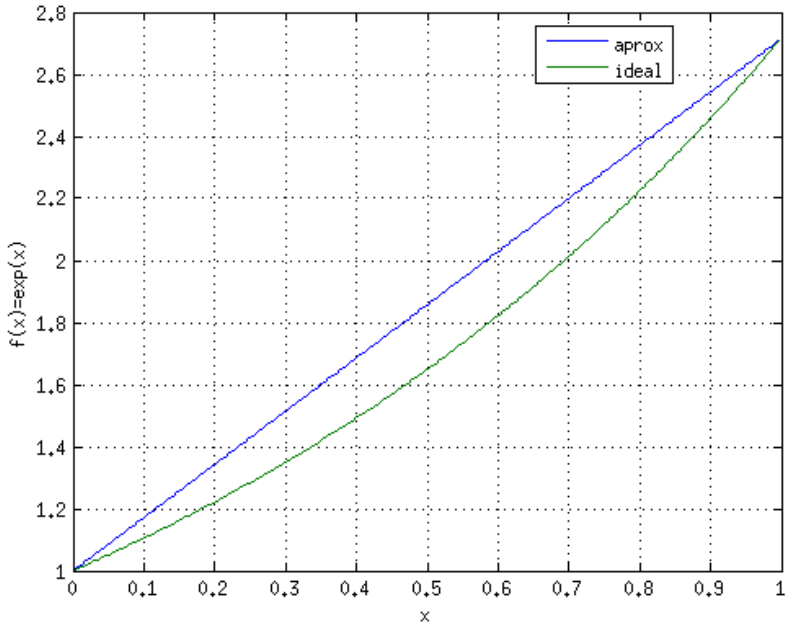


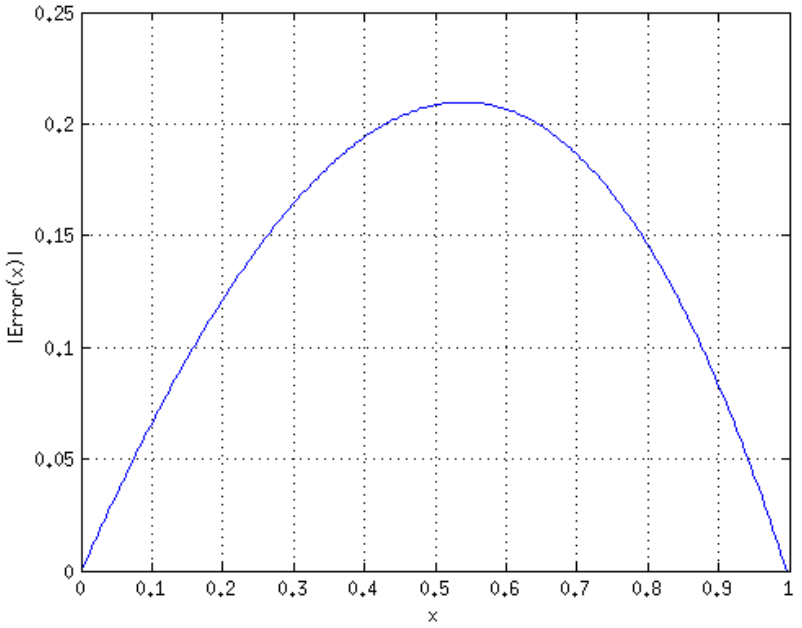
Métodos basados en tablas

- **Interpolación lineal**: $y=e^x$ con x en [0,1[

  [Aproximación en $[x_0,x_1] \Rightarrow y_a=m\cdot\Delta x+c$'

Exactitud del cálculo: 2 bits





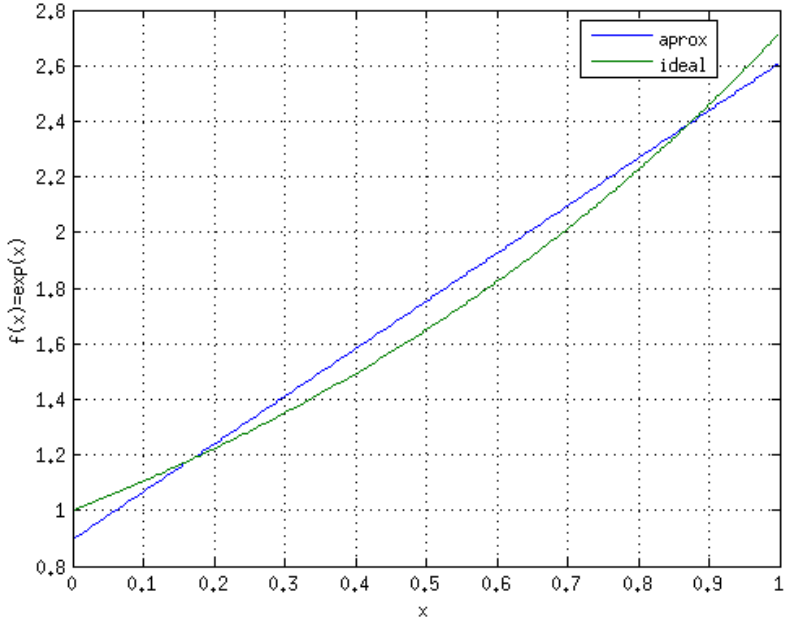
Métodos basados en tablas

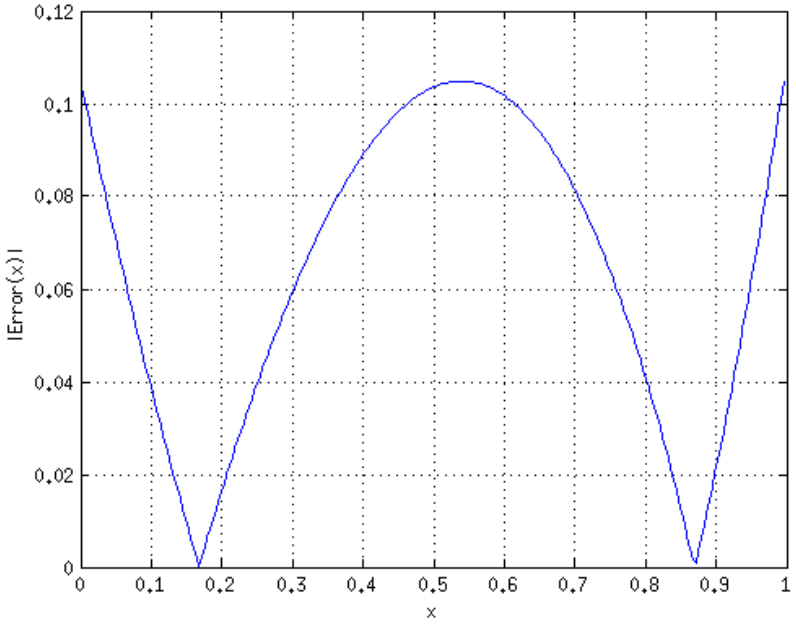
- **Interpolación lineal**: $y=e^x$ con x en [0,1[

  [Aproximación en $[x_0,x_1] \Rightarrow y_a=m\cdot\Delta x+c'$ con $c'=c+offset$

  $offset=-max\_error/2$

Exactitud del cálculo: 3 bits





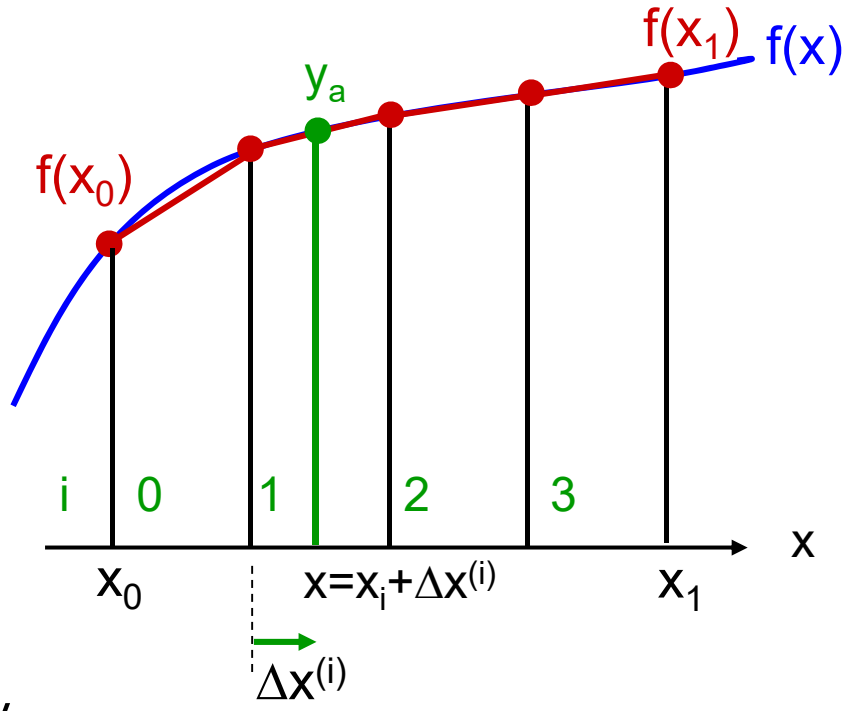
Métodos basados en tablas

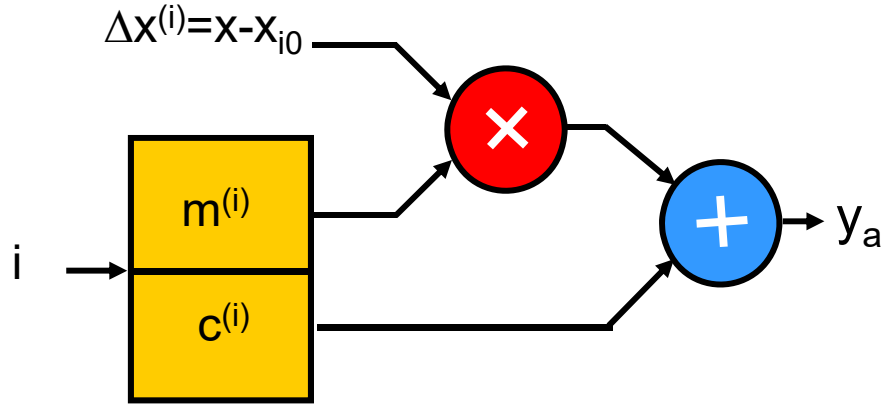
- **Interpolación lineal con división del intervalo**: y=f(x)

  Aproximación en $[x_0,x_1] \Rightarrow y_a^{(i)}=m^{(i)}\cdot\Delta x+c^{(i)}$, siendo i el índice del intervalo

  $m=(f(x_{i1})-f(x_{i0}))/(x_{i1}-x_{i0})$

  $c=f(x_{i0})+offset\_i$





Métodos basados en tablas

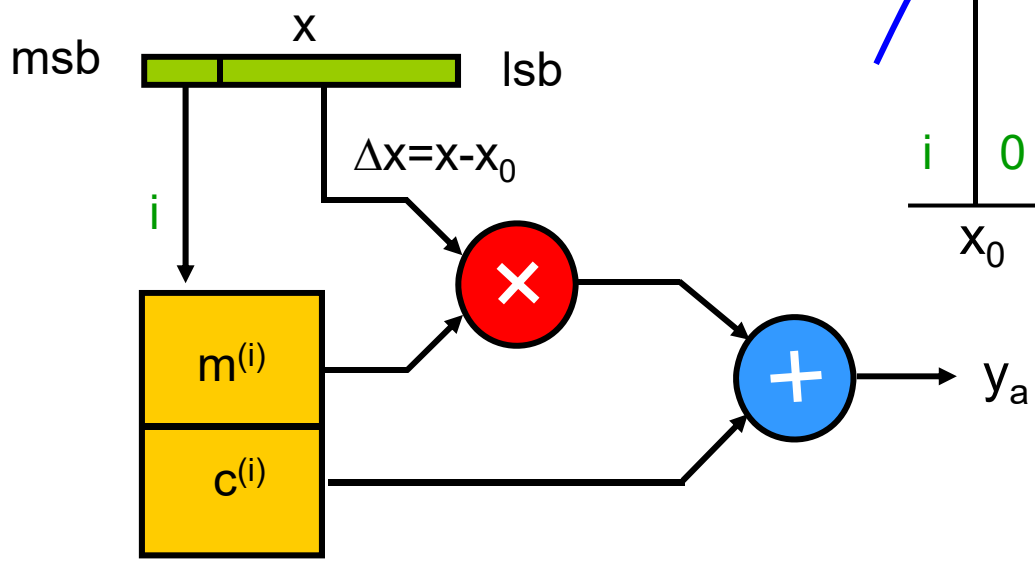
- **Interpolación lineal con división del intervalo**: y=f(x)

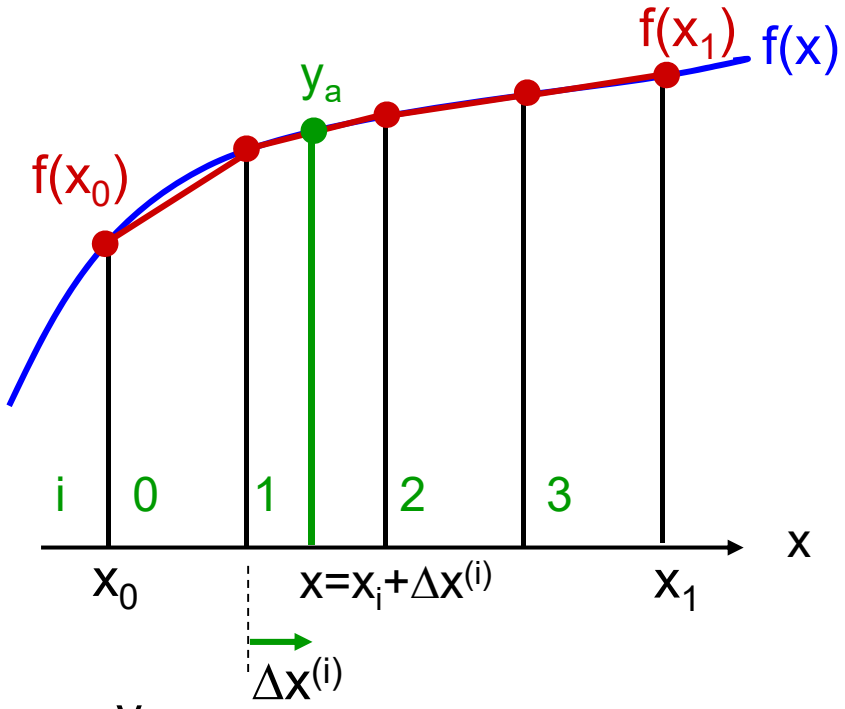
  Aproximación en $[x_0,x_1] \Rightarrow y_a^{(i)}=m^{(i)}\cdot\Delta x+c^{(i)}$, siendo i el índice del intervalo

  $m=(f(x_{i1})-f(x_{i0}))/(x_{i1}-x_{i0})$

  $c=f(x_{i0})$

Si $(x_{i1}-x_{i0})=2^a$ se simplifica la implementación hardware





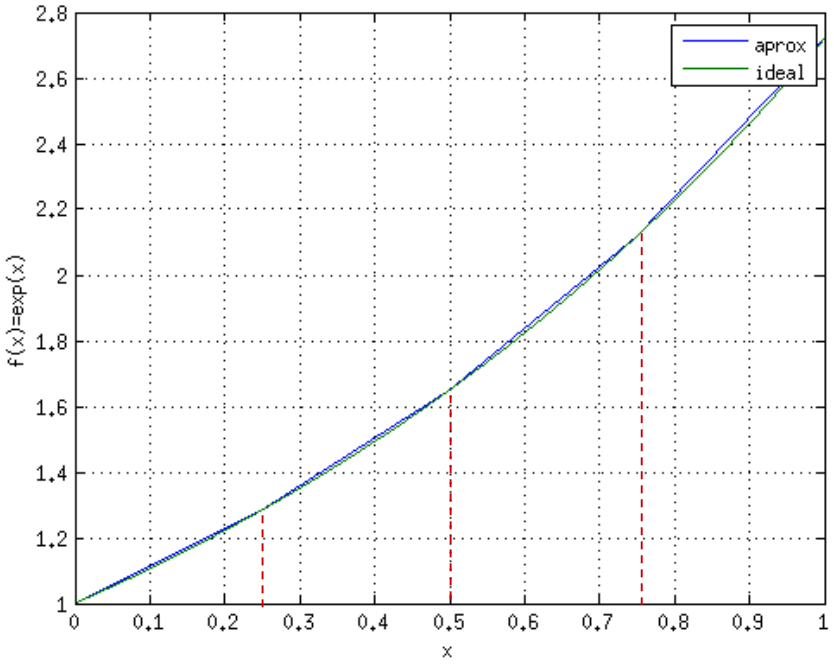
Métodos basados en tablas

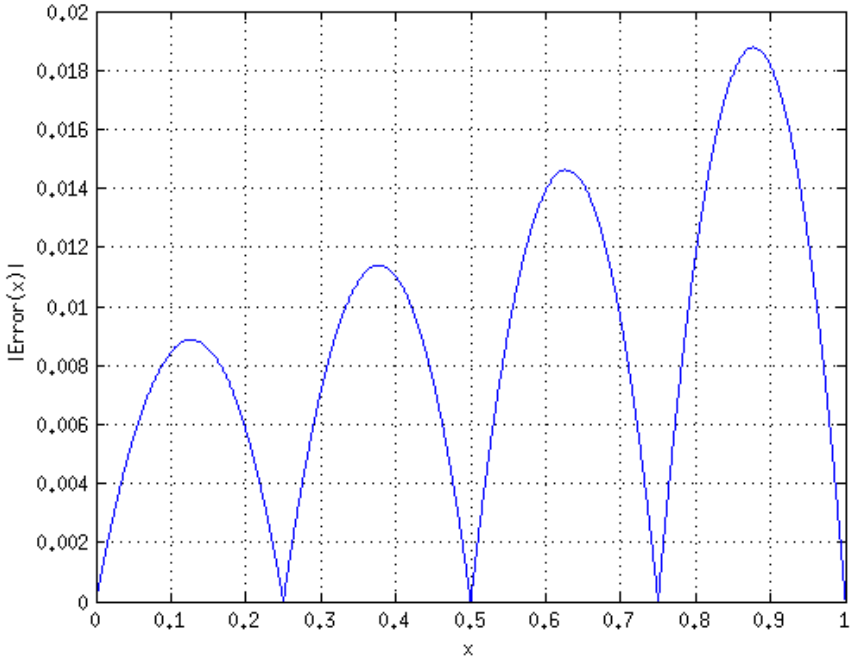
- **Interpolación lineal con división del intervalo**: $y=e^x$ con x en [0,1[

  Aproximación en $[x_0,x_1] \Rightarrow y_a^{(i)}=m^{(i)}\cdot\Delta x+c^{(i)}$

  División del intervalo en 4 zonas con $(x_{i1}-x_{i0}) =2^{-2}$

Exactitud del cálculo: 5 bits





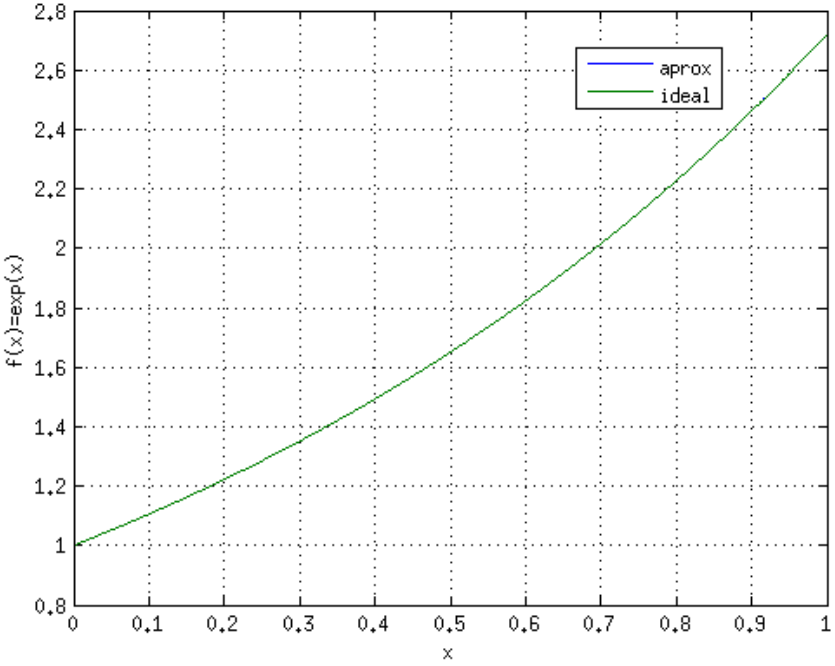
Métodos basados en tablas

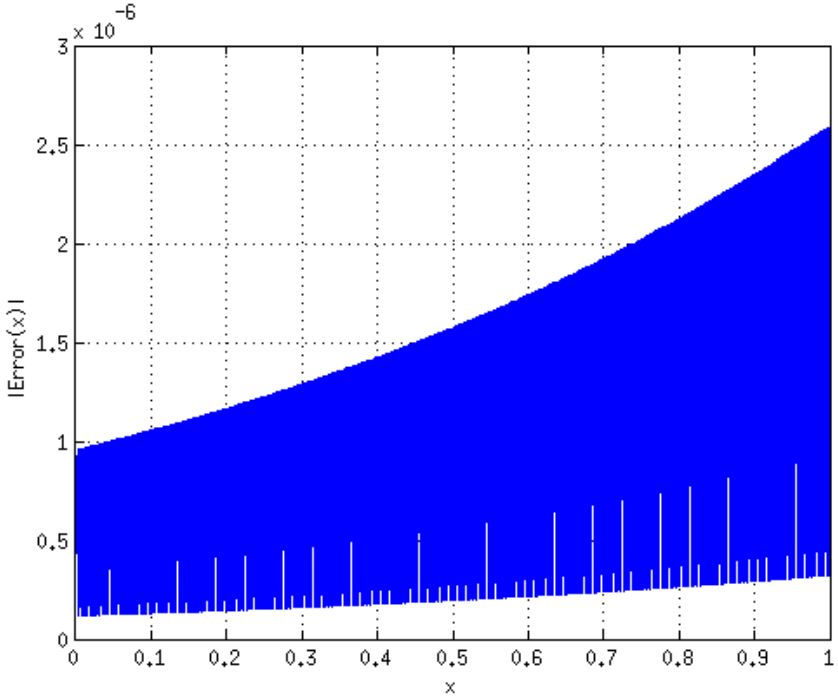
- **Interpolación lineal con división del intervalo**: $y=e^x$ con x en [0,1[

  División del intervalo en 256 zonas con $(x_{i1}-x_{i0}) = 2^{-9}$

Recursos HW: 1 M9K (256x36 bits)
1 mult
1 sumador

Exactitud del cálculo: 18 bits





Métodos iterativos

Método de Newton-Raphson : x=1/d

x_0=LUT(1/d)
x_{k+1}=x_k(2-x_kd)

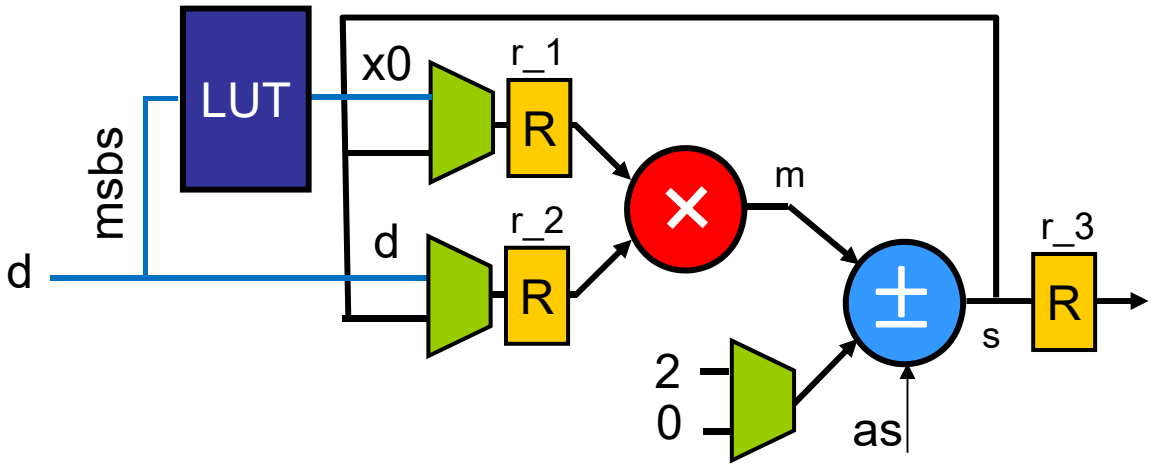
$$x_{k+1} = x_k \cdot (2 - x_k \cdot d)$$

dependencias en operaciones

Implementación con 1 mult

2 ciclos clk por iteración

ciclos clk	operaciones
1º	$v_k = (2 - x_k \cdot d)$
2º	$x_{k+1} = x_k \cdot v_k$



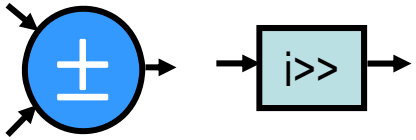
$$Thr = \frac{f_{clk}}{2 \cdot \#it}$$

Métodos iterativos

Algoritmo CORDIC

$$X_{i+1} = X_i - d_i 2^{-i} Y_i$$
$$Y_{i+1} = Y_i + d_i 2^{-i} X_i$$
$$Z_{i+1} = Z_i - d_i \tan^{-1}(2^{-i})$$

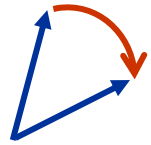
Operadores aritméticos:

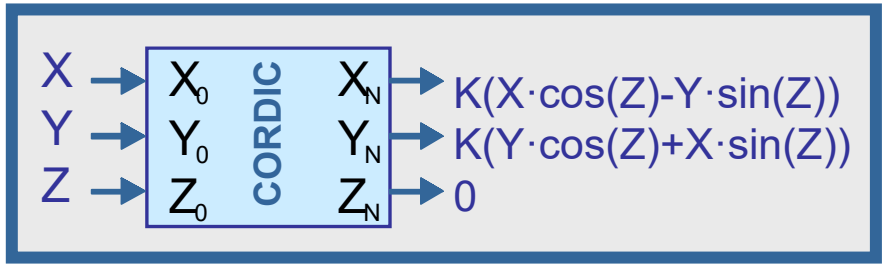


- Rango de convergencia: $\pm\pi/2$
- P+1 iteraciones para precisión P
- Dimensionado de los operadores: 2+P+log2(P) bits

Modo Rotación

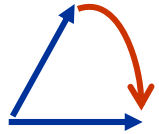
$$d_i = sign(Z_i)$$

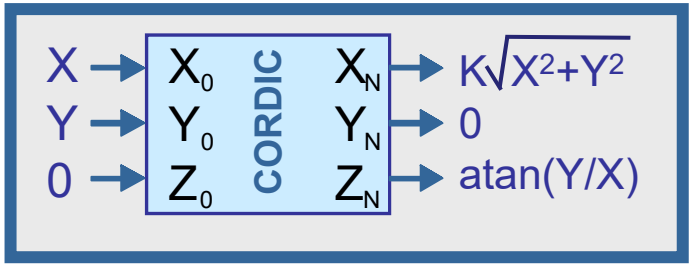




Modo Vectorización

$$d_i = -sign(Y_i)$$





K=1.6468

Evaluación de polinomios

Func	Polynomial approximation	Conditions
$1/x$	$1 + y + y^2 + y^3 + \cdots + y^i + \cdots$	$0<x<2,\ y=1-x$
e^x	$1 + x/1! + x^2/2! + x^3/3! + \cdots + x^i/i! + \cdots$	
$\ln x$	$-y - y^2/2 - y^3/3 - y^4/4 - \cdots - y^i/i - \cdots$	$0<x \quad 2,\ y=1-x$
$\ln x$	$2[z + z^3/3 + z^5/5 + \cdots + z^{2i+1}/(2i+1) + \cdots]$	$x>0,\ z=\frac{x-1}{x+1}$
$\sin x$	$x - x^3/3! + x^5/5! - x^7/7! + \cdots + (-1)^i x^{2i+1}/(2i+1)! + \cdots$	
$\cos x$	$1 - x^2/2! + x^4/4! - x^6/6! + \cdots + (-1)^i x^{2i}/(2i)! + \cdots$	
$\tan^{-1} x$	$x - x^3/3 + x^5/5 - x^7/7 + \cdots + (-1)^i x^{2i+1}/(2i+1) + \cdots$	$-1 < x < 1$
$\sinh x$	$x + x^3/3! + x^5/5! + x^7/7! + \cdots + x^{2i+1}/(2i+1)! + \cdots$	
$\cosh x$	$1 + x^2/2! + x^4/4! + x^6/6! + \cdots + x^{2i}/(2i)! + \cdots$	
$\tanh^{-1} x$	$x + x^3/3 + x^5/5 + x^7/7 + \cdots + x^{2i+1}/(2i+1) + \cdots$	$-1 < x < 1$

Evaluación de polinomios

Método de Horner:

$$
\begin{aligned}
f(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_{n-1} x^{n-1} + a_n x^n = \\
&= a_0 + x(a_1 + a_2 x^1 + a_3 x^2 + \cdots + a_{n-1} x^{n-2} + a_n x^{n-1}) = \\
&= a_0 + x(a_1 + x(a_2 + a_3 x^1 + \cdots + a_{n-1} x^{n-3} + a_n x^{n-2})) = \\
&= \cdots = \\
&= a_0 + x(a_1 + x(a_2 + x(a_3 + x(\cdots + x(a_{n-1} + a_n x)\cdots))))
\end{aligned}
$$

Se necesitan **n** operaciones $x \cdot v_{i-1} + a_i$ para evaluar un polinomio de orden **n**

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + x(a_1 + x(a_2 + x a_3))$$

dependencias en operaciones

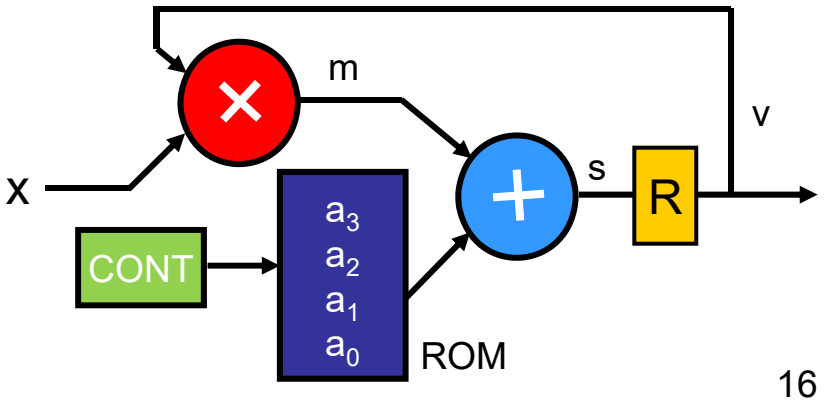
Operaciones en cada ciclo de reloj:

1º $v_1 = a_3$

2º $v_2 = a_2 + x \cdot v_1$

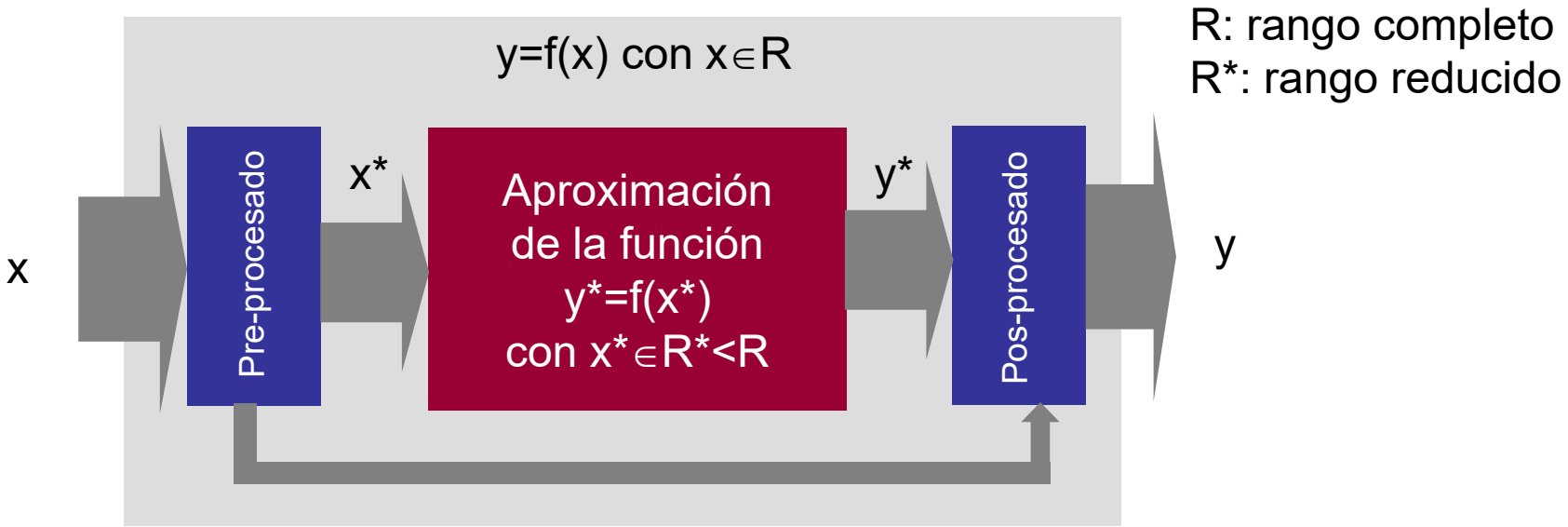
3º $v_3 = a_1 + x \cdot v_2$

4º $v_4 = a_0 + x \cdot v_3$



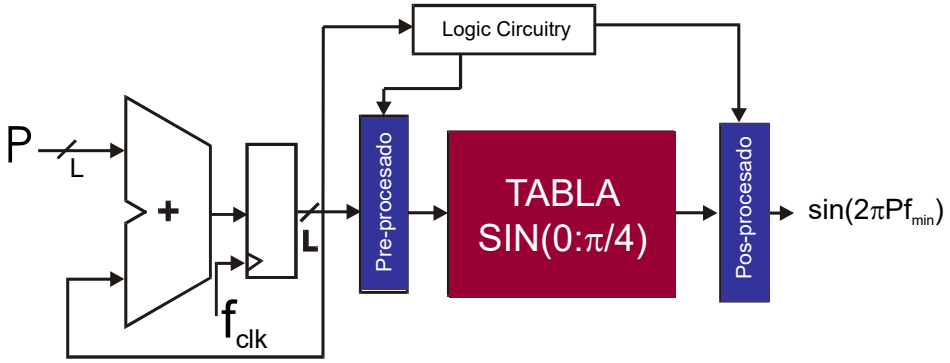
Reducción-ampliación del rango

- No suele ser eficiente aproximar la función en todo el rango de aplicación
- Algunos métodos solo convergen para entradas en un rango reducido
- Método general de aproximación de funciones:
  - **Preprocesado**: reducción del rango
  - **Aproximación** de la función en el rango reducido
  - **Pos-procesado**: reconstrucción del resultado, expansión al rango completo



Simetría de cuarto de onda

Se almacena sólo un cuarto de onda, el resto se obtiene por simetría vertical (SV) u horizontal (SH)



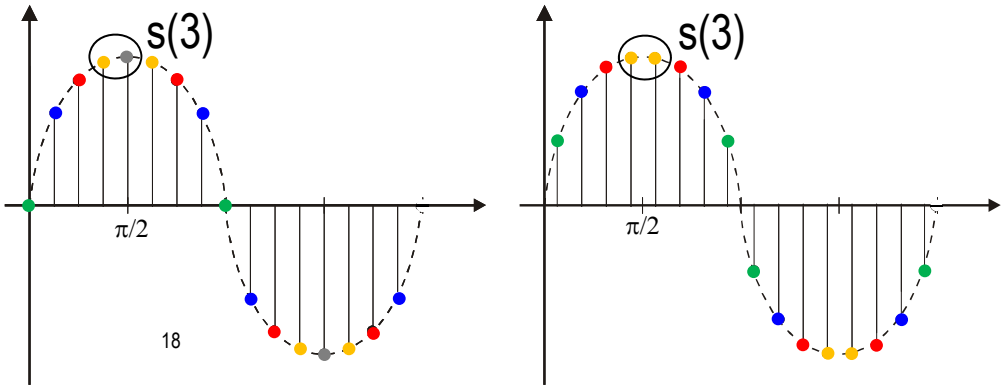
	fase	d_3	d_2	d_1	d_0	$s(d_1,d_0)$
I	0	0	0	0	0	s(0)
	π/8	0	0	0	1	s(1)
	π/4	0	0	1	0	s(2)
	3π/8	0	0	1	1	s(3)
II	π/2	0	1	0	0	s(3)
	5π/8	0	1	0	1	s(2)
	3π/4	0	1	1	0	s(1)
	7π/8	0	1	1	1	s(0)
III	π	1	0	0	0	-s(0)
	9π/8	1	0	0	1	-s(1)
	5π/4	1	0	1	0	-s(2)
	11π/8	1	0	1	1	-s(3)
IV	3π/2	1	1	0	0	-s(3)
	13π/8	1	1	0	1	-s(2)
	7π/4	1	1	1	0	-s(1)
	15π/8	1	1	1	1	-s(0)

La memoria se direcciona con **L-2 bits**
Los dos bits más significativos se utilizan para controlar la simetría:

SV: $d_3 \Rightarrow$ complementa a dos la salida de la tabla

SH: $d_2 \Rightarrow$ complementa a uno las líneas de direcciones

Hay que muestrear en medio del intervalo de muestreo para mantener la simetría



Reducción-ampliación del rango

Reducción de rango aditiva:

x: variable con rango completo

x*: variable con rango reducido

$x^* = x - k \cdot C$, con C=cte y k entero

Ejemplo: $y=e^x$ con $x \in [0,16[$

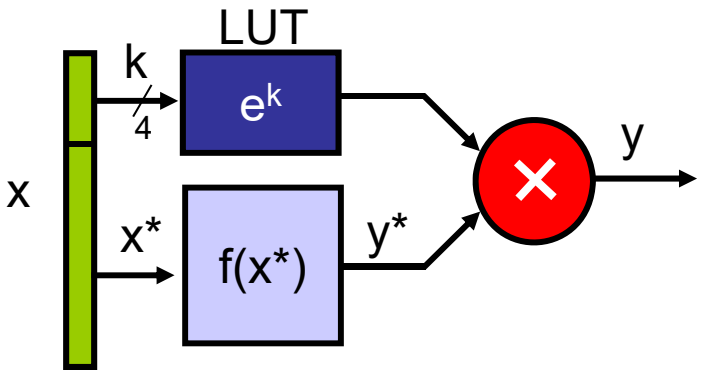
$x^*=x-k \cdot C=x-k$ con C=1 para que $x^* \in [0,1[$ $\Leftarrow$ x^* es la parte fraccional de x

k es la parte entera

$$y= e^x= e^{x^*+k} = e^{x^*} \cdot e^k = y^* \cdot e^k$$

Procedimiento:

- Tabular e^k en una pequeña tabla
- Calcular y*
- Multiplicar resultados de ambos



Reducción-ampliación del rango

Reducción de rango multiplicativa:

x: variable con rango completo

x*: variable con rango reducido

$x^* = x/C^k$, con C=cte y k entero

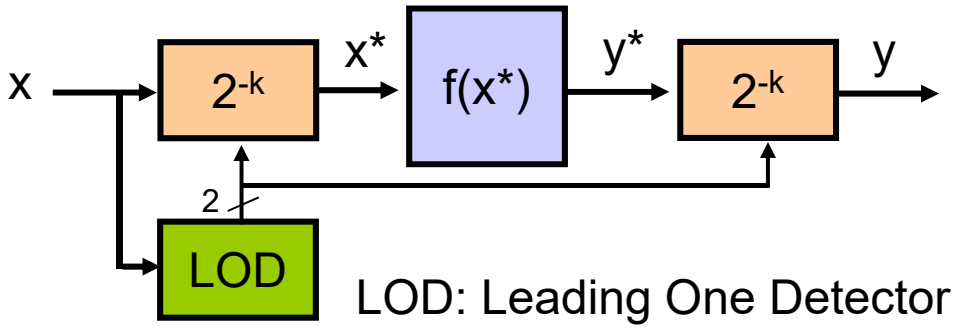
Ejemplo: y=1/x con $x \in]0,16[$

$x^* = x/C^k = x/2^k$ con C=2 para que $x^* \in [0,1[$ ⇐ la entrada x se desplaza k veces a la derecha

$$y = 1/x = 1/(x^*2^k) = (1/x^*)\cdot 2^{-k} = y^*\cdot 2^{-k}$$

Procedimiento:

- Obtener x* (k desplazamientos)
- Calcular y*=1/x*
- Desplazar el resultado k bits



LOD: Leading One Detector

Bibliografía

- **Elementary functions: Algoritms ans Implementation**,
  Jean-Michel Muller,
  Ed. Birhauser

- **Computer Arithmetic: Algorithms ans Hardware Design**,
  Behrooz Parhami,
  Ed. Oxford University Press