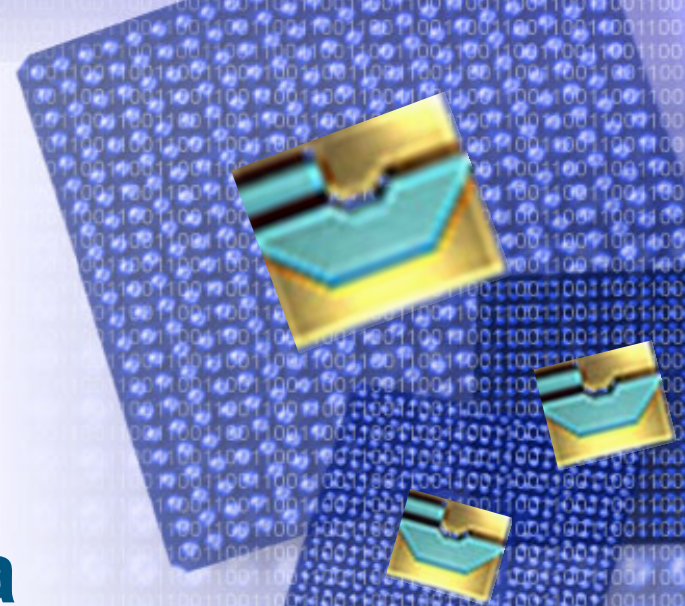




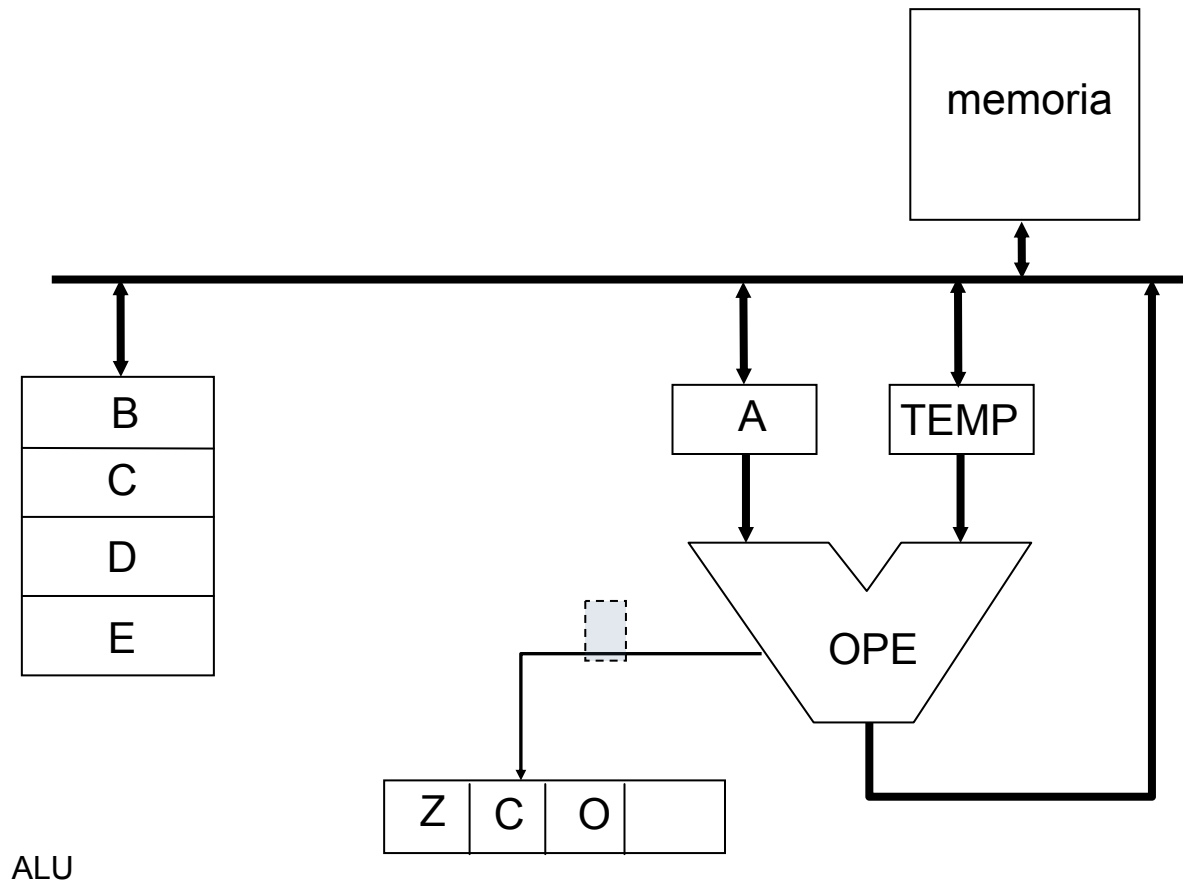
Unidad Aritmético Lógica





Introducción

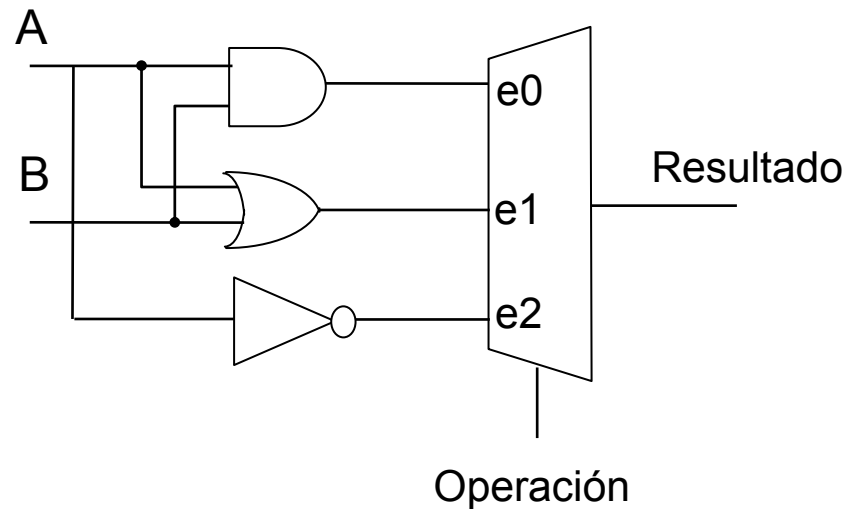
- Operador aritmético y lógico (uno o varios).
- El Acumulador.
- Uno o varios registros temporales.
- Un banco de registros.
- Indicadores de resultado:
 - Acarreo (C)
 - Negativo (N)
 - Desbordamiento (O)
 - Cero (Z)





Operaciones lógicas

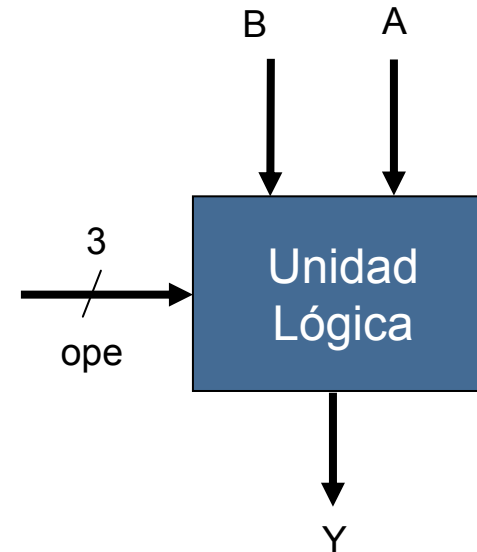
- Fáciles de implementar \Rightarrow Correspondencia directa con Hardware.
- Puertas lógicas AND, OR, OR-EXCLUSIVA, INVERSORES,...





Unidad Lógica

ope	función
000	NOT A
001	NOT B
010	A AND B
011	A OR B
100	A NAND B
101	A NOR B
110	A XOR B
111	A NO_XOR B





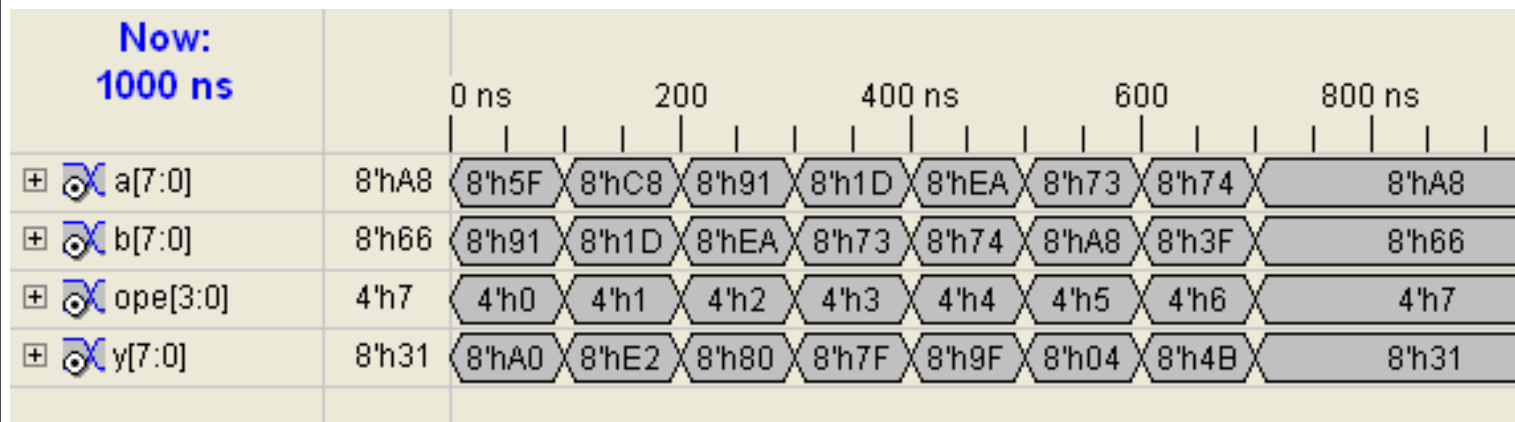
Unidad Lógica

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
(generic n:integer:=8);
entity unidad_logica is
    Port ( a : in std_logic_vector(n-1 downto 0);
          b : in std_logic_vector(n-1 downto 0);
          ope : in std_logic_vector(3 downto 0);
          y : out std_logic_vector(n-1 downto 0));
end unidad_logica;
architecture comporta of unidad_logica is
begin
    PROCESS(a, b, ope)
    begin
        CASE ope(2 DOWNT0 0) IS           -- puede ser cualquier bit
        WHEN "000" => y <= NOT a;
        WHEN "001" => y <= NOT b;
        WHEN "010" => y <= a AND b;
        WHEN "011" => y <=a OR b;
        WHEN "100" => y <=a NAND b;
        WHEN "101" => y <=a NOR b;
        WHEN "110" => y <=a XOR b;
        WHEN OTHERS => y <= NOT (a XOR b);
        END CASE;
    end process;
end comporta;
```



Unidad Lógica

ope	función
000	NOT A
001	NOT B
010	A AND B
011	A OR B
100	A NAND B
101	A NOR B
110	A XOR B
111	A NO_XOR B





Operaciones de desplazamiento

- Consisten en trasladar los bits de una palabra hacia la izquierda o derecha.
- Si llamamos O al operando origen, de n bits ($o_{n-1} \dots o_2 o_1 o_0$) y D al operando destino, de n bits, ($d_{n-1} \dots d_2 d_1 d_0$)

$$d_{i+k} = o_i \text{ para } i=0,1,\dots,n-1$$

Donde k, indica el número de desplazamientos y el signo el sentido de los mismos:

izquierda el signo es más (+)

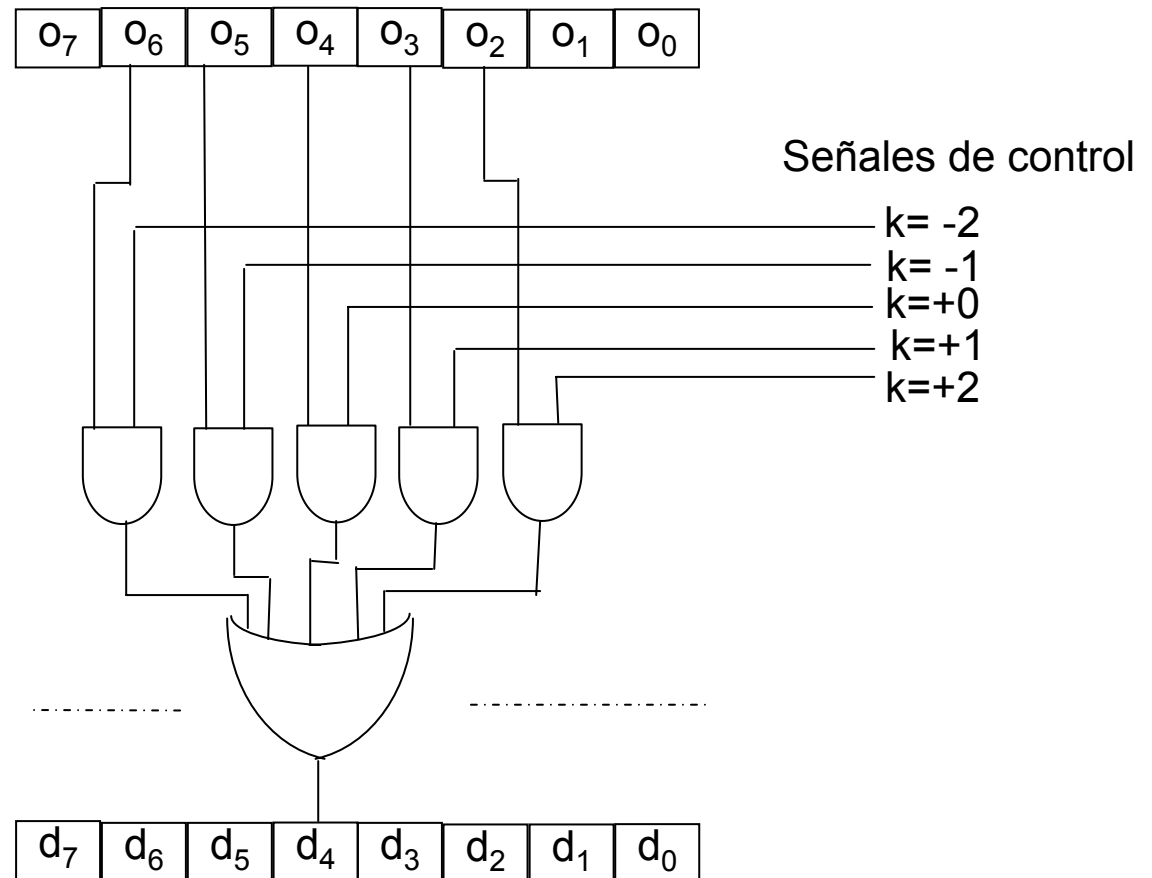
derecha el signo es menos (-)

La cantidad de desplazamientos depende de la complejidad de las máquinas, las más sencilla admiten $k=1$ y $k=-1$.



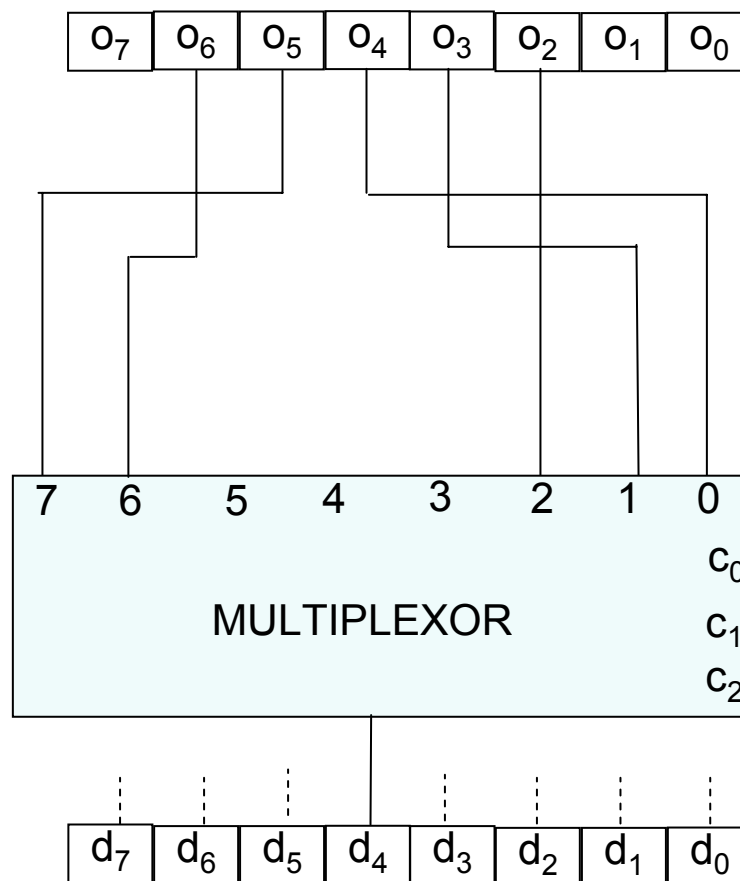
Operaciones de desplazamiento

- La complejidad es elevada.
- Las señales de control son las mismas para cada bit.
- Las puertas pueden sustituirse por multiplexores
- Dependiendo de cómo se traten los extremos, se obtienen tres tipos de desplazamientos:
 - Lógicos
 - Circulares
 - Aritméticos





Operaciones de desplazamiento



k expresado en C2

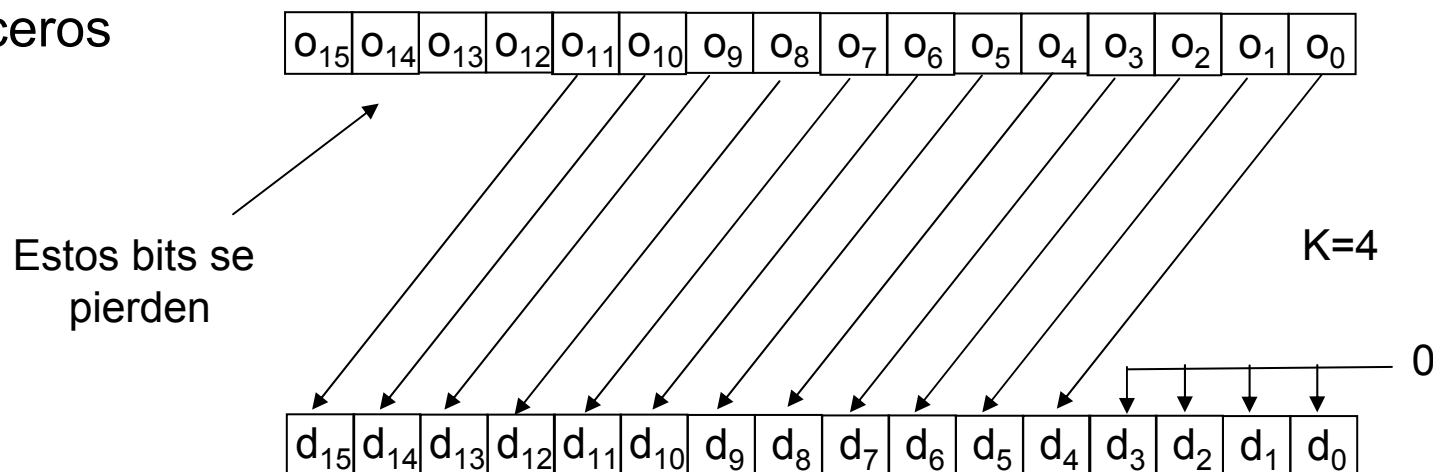
Señales de control

0	1	0	1	0
1	0	0	1	1
0	0	0	1	1
$k=+2$	$k=+1$	$k=+0$	$k=-1$	$k=-2$

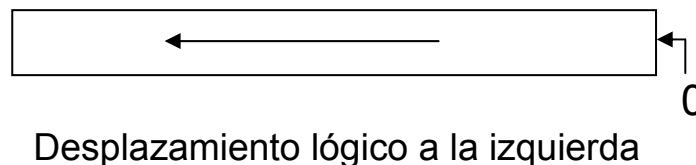
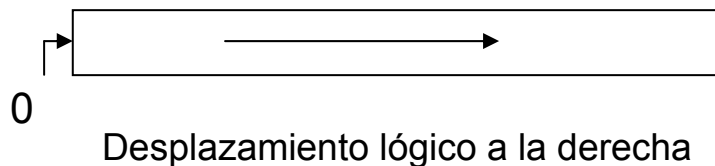


Desplazamientos lógicos

- Los valores extremos se completan con ceros, aunque se pueden plantear desplazamientos lógicos con inclusión de unos en lugar de ceros

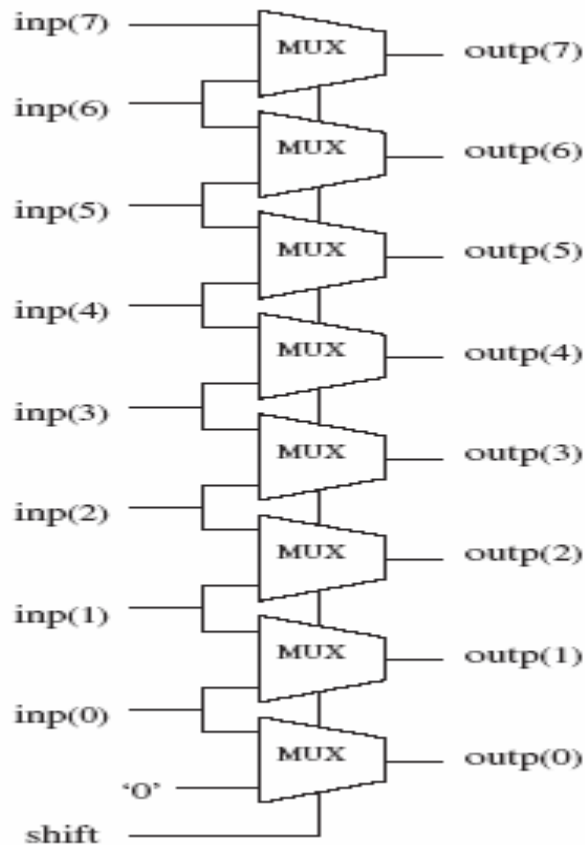


- Habitualmente, el origen y destino es la misma palabra.





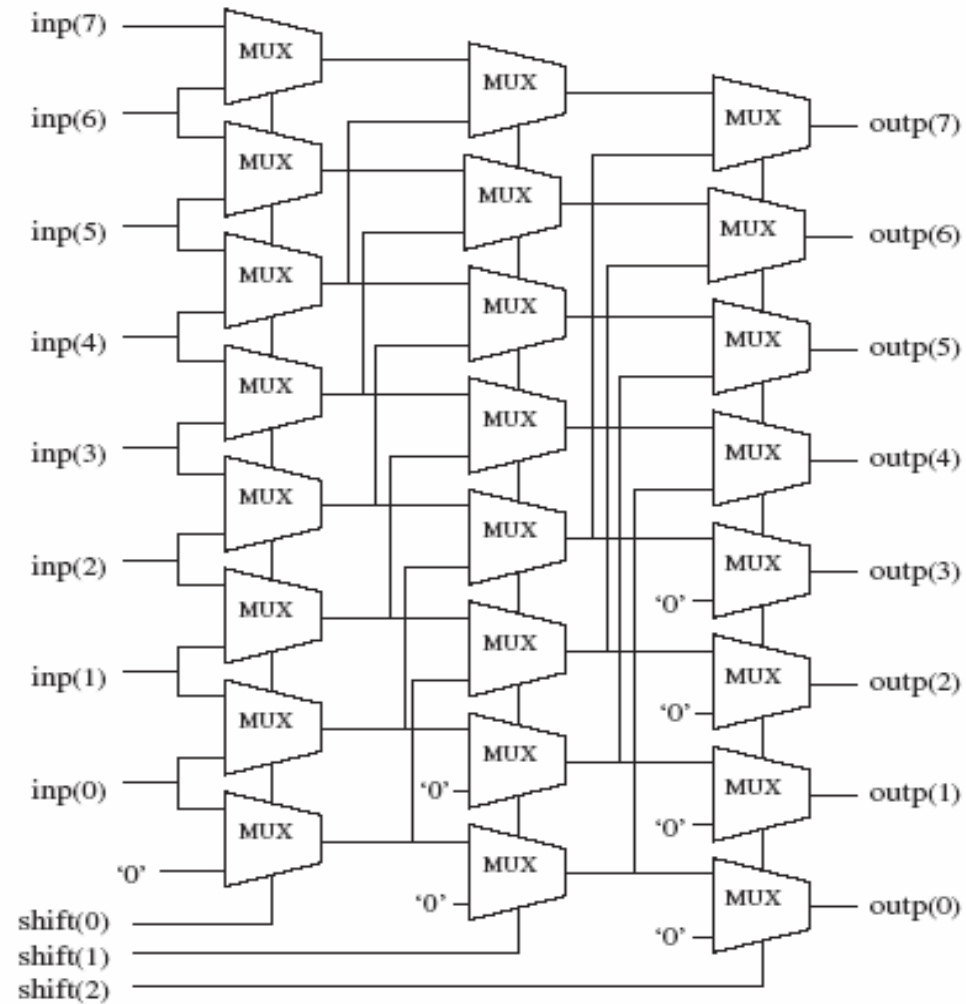
Operaciones de desplazamiento



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity shifter is
    GENERIC (N: INTEGER:=8);
    Port ( ENTRADA : in std_logic_vector(N-1 downto 0);
          SALIDA : out std_logic_vector(N-1 downto 0);
          shift : in std_logic);
end shifter;
architecture rtl of shifter is
begin
    process (ENTRADA, shift)
    begin
        if (shift='0') then
            SALIDA <= ENTRADA;
        else
            SALIDA(0) <= '0';
            for i in 1 to ENTRADA'high loop
                SALIDA(i) <= ENTRADA(i-1);
            end loop;
        end if;
    end process;
end rtl;
```



Operaciones de desplazamiento





Operaciones de desplazamiento

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity shifter2 is
    Port ( inp : in std_logic_vector(7 downto 0);
          shift : in std_logic_vector(2 downto 0);
          outp : out std_logic_vector(7 downto 0));
end shifter2;
architecture Behavioral of shifter2 is
begin
    PROCESS (inp, shift)
        VARIABLE temp1: STD_LOGIC_VECTOR (7 DOWNT0 0);
        VARIABLE temp2: STD_LOGIC_VECTOR (7 DOWNT0 0);
    BEGIN
        ---- 1st shifter ----
        IF (shift(0)='0') THEN
            temp1 := inp;
        ELSE
            temp1(0) := '0';
            FOR i IN 1 TO inp'HIGH LOOP
                temp1(i) := inp(i-1);
            END LOOP;
        END IF;
    END PROCESS;
```



Operaciones de desplazamiento

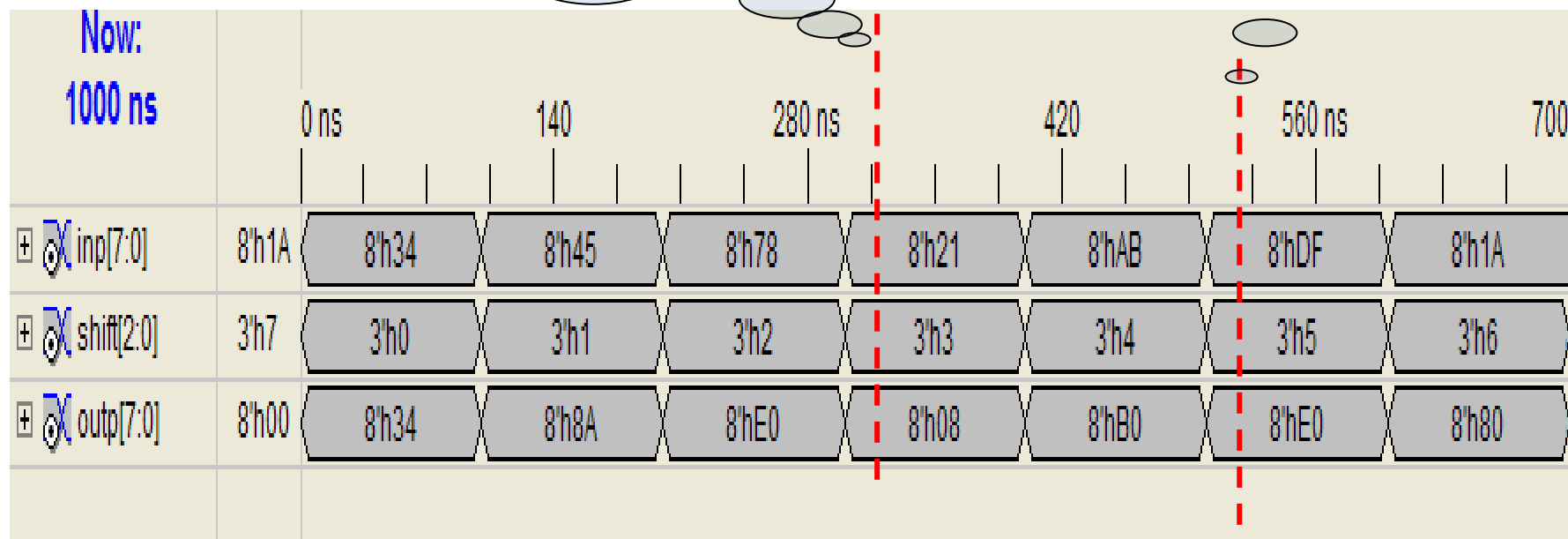
```
---- 2nd shifter ----
  IF (shift(1)='0') THEN
    temp2 := temp1;
  ELSE
    FOR i IN 0 TO 1 LOOP
      temp2(i) := '0';
    END LOOP;
    FOR i IN 2 TO inp'HIGH LOOP
      temp2(i) := temp1(i-2);
    END LOOP;
  END IF;
---- 3rd shifter ----
  IF (shift(2)='0') THEN
    outp <= temp2;
  ELSE
    FOR i IN 0 TO 3 LOOP
      outp(i) <= '0';
    END LOOP;
    FOR i IN 4 TO inp'HIGH LOOP
      outp(i) <= temp2(i-4);
    END LOOP;
  END IF;
END PROCESS;
end Behavioral;
```



Operaciones de desplazamiento

Cuidado, primero desplaza 1
Después desplaza 2

Cuidado, primero desplaza 1
Después desplaza 4

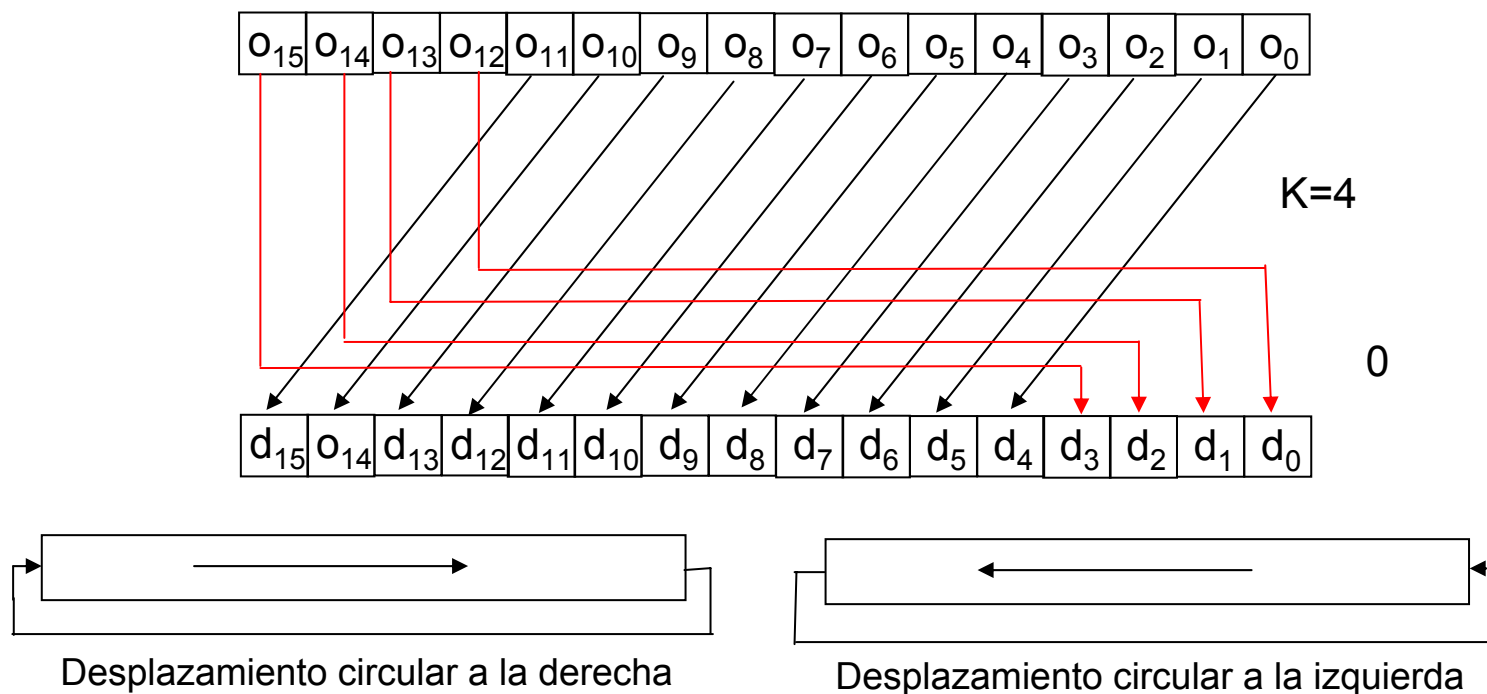




Desplazamientos circulares

- Los bits del origen que sobran por un lado, se insertan en el destino por el otro, matemáticamente:

$$o_i = d_{(n+i+k) \bmod n} \text{ para } i=1, 2, \dots, n-1$$





Operaciones aritméticas: Suma y resta

- ❶ La suma se utiliza como operación primitiva para procesar muchas funciones aritméticas, por lo tanto merece una atención particular.
- ❷ El algoritmo clásico de lápiz y papel implica un procesamiento secuencial de los acarreos, cada uno de ellos depende de los que le preceden.
- ❸ El tiempo de procesamiento, por lo tanto, depende del número n de dígitos del operando.
- ❹ Para minimizar el tiempo de procesamiento, vamos a ver varios métodos.



Suma de números naturales

Algoritmo básico:

Considerando la base de representación **B** de dos números de **n** dígitos

$$x = x_{n-1} \cdot B^{n-1} + x_{n-2} \cdot B^{n-2} + \dots + x_0 \cdot B^0$$

$$y = y_{n-1} \cdot B^{n-1} + y_{n-2} \cdot B^{n-2} + \dots + y_0 \cdot B^0$$

la suma $z = x + y + c_{in}$ procesa $n+1$ dígitos

```
c(0)=c_in
for i in 0 to n-1 loop
  if x(i)+y(i)+c(i)>B-1 then
    c(i+1):=1;
  else
    c(i+1):=0;
  end if;
  z(i):=(x(i)+y(i)+c(i)) mod B;
end loop;
z(n):=c(n)
```

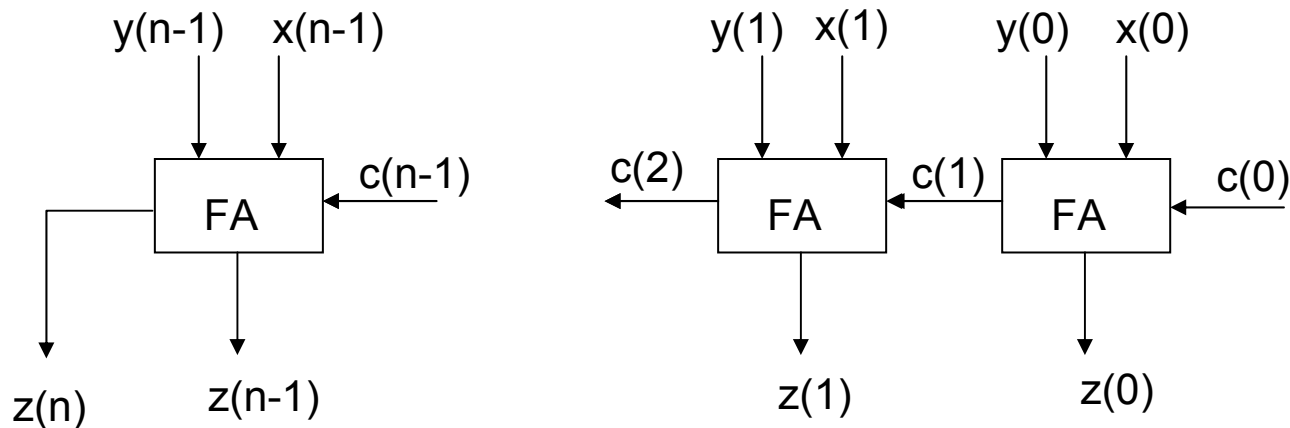
Como $c(i+1)$ es una función de $c(i)$ el tiempo de ejecución del algoritmo 1, es proporcional a **n**

Algoritmo 1
lápiz y papel



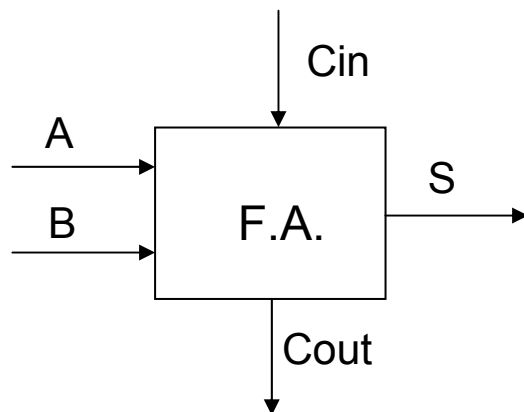
Sumador con propagación de acarreo

- La estructura para sumar dos números de n bits es colocar en cascada n sumadores completos.
- El acarreo se propaga de una etapa a la siguiente: Sumador con Propagación de Acarreo (Carry Propagated Adder)





Sumador completo (F.A.)



Entradas			Salidas	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = A \oplus B \oplus Cin$$

$$Cout = A \cdot B + A \cdot Cin + B \cdot Cin$$



Sumador con propagación de acarreo

- Si llamamos C_{FA} y T_{FA} al coste y tiempo computacional de un FA (sumador completo), para un sumador de n bits tendremos:

$$C_{\text{sumador_básico}}(n) = n \cdot C_{FA}$$
$$T_{\text{sumador_básico}}(n) = n \cdot T_{FA}$$

- El comportamiento del sumador de acarreo propagado será:

```
c(0)=c_in
for i in 0 to n-1 generate

    c(i+1)=x(i) · y(i) + x(i)·c(i) + y(i)·c(i)

    z(i) =x(i) xor y(i) xor c(i)

end generate;
z(n):=c(n)
```





Sumador con anticipación de acarreo

Carry Chain Adder (CCA)

- Se puede acelerar el proceso de suma si se tiene en cuenta que se puede obtener acarreo de acuerdo a dos condiciones

Señal generadora de acarreo : $G_i = a_i + b_i$

Señal propagadora de acarreo: $P_i = a_i \cdot b_i$

El acarreo de la etapa i: $C_i = G_i + P_i \cdot C_{i-1}$

- Si particularizamos para 4 bits:

$$C_0 = G_0 + P_0 \cdot C_{-1}$$

$$C_1 = G_1 + P_1 \cdot C_0$$

$$C_2 = G_2 + P_2 \cdot C_1$$

$$C_3 = G_3 + P_3 \cdot C_2$$



Sumador con anticipación de acarreo Carry Chain Adder (CCA)

- Desarrollando las expresiones y poniéndolas en función de C_{-1} :

$$C_0 = G_0 + P_0 \cdot C_{-1}$$

$$C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_{-1}$$

$$C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_{-1}$$

$$C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_{-1}$$

- Todos los acarreos dependen de a_i y b_i .
- Estas expresiones se resuelven como suma de productos.
- Tres niveles de puertas lógicas para obtener cada uno de los acarreos.



Sumador con anticipación de acarreo Carry Chain Adder (CCA)

• Por lo tanto el acarreo siguiente se puede calcular como

```
if p[x(i),y(i)]=1 then  
    c(i+1):=c(i);  
else  
    c(i+1):=g[x(i),y(i)];  
end if;
```




Sumador con anticipación de acarreo Carry Chain Adder (CCA)

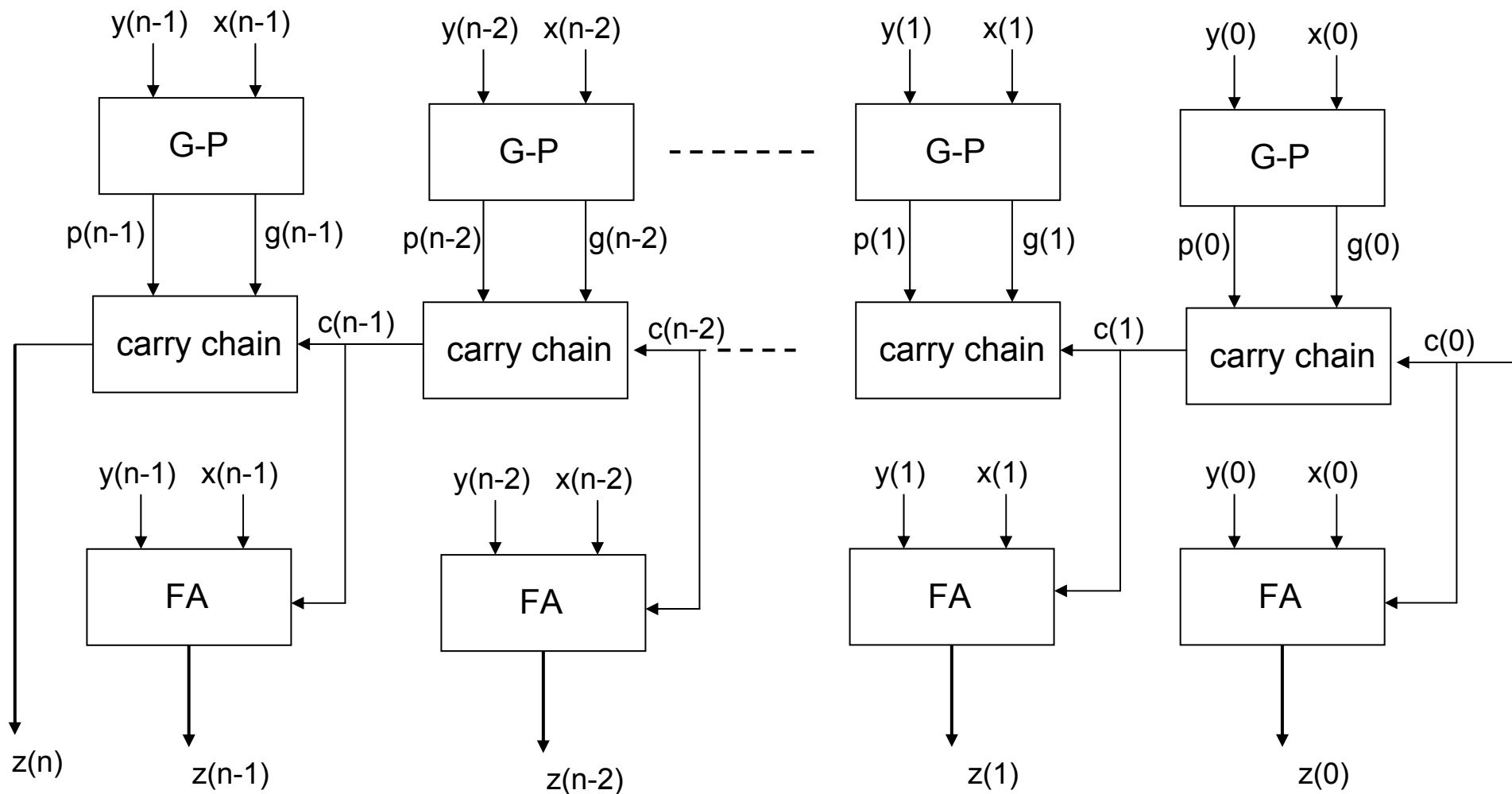
```
-- cálculo de generación y propagación
for i in 0 to n-1 loop
    g(i):= g[x(i),y(i)];
    p(i):= p[x(i),y(i)];
end loop;
-- cálculo del acarreo
c(0)=c_in
for i in 0 to n-1 loop
    if p(i)=1 then
        c(i+1):=c(i);
    else
        c(i+1):=g(i);
    end if;
end loop;
-- cálculo de la suma
for i in 0 to n-1 loop
    z(i):=(x(i)+y(i)+c(i)) mod B;
end loop;
z(n):=c(n)
```

Algoritmo 2
acarreo anticipado



Sumador con anticipación de acarreo

Carry Chain Adder (CCA)



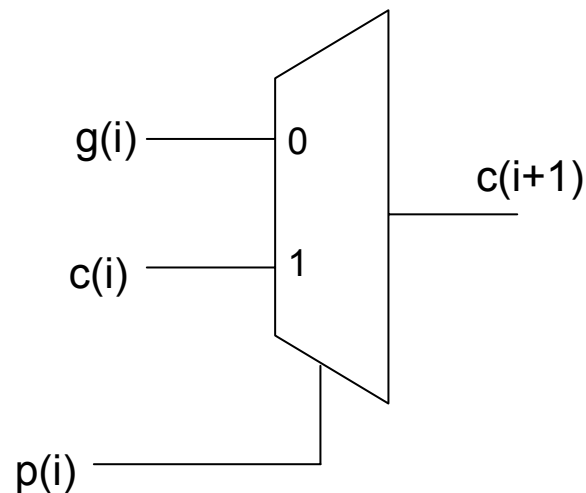


Sumador con anticipación de acarreo

Carry Chain Adder (CCA)

El bloque carry chain calcula el acarreo siguiente, es decir,

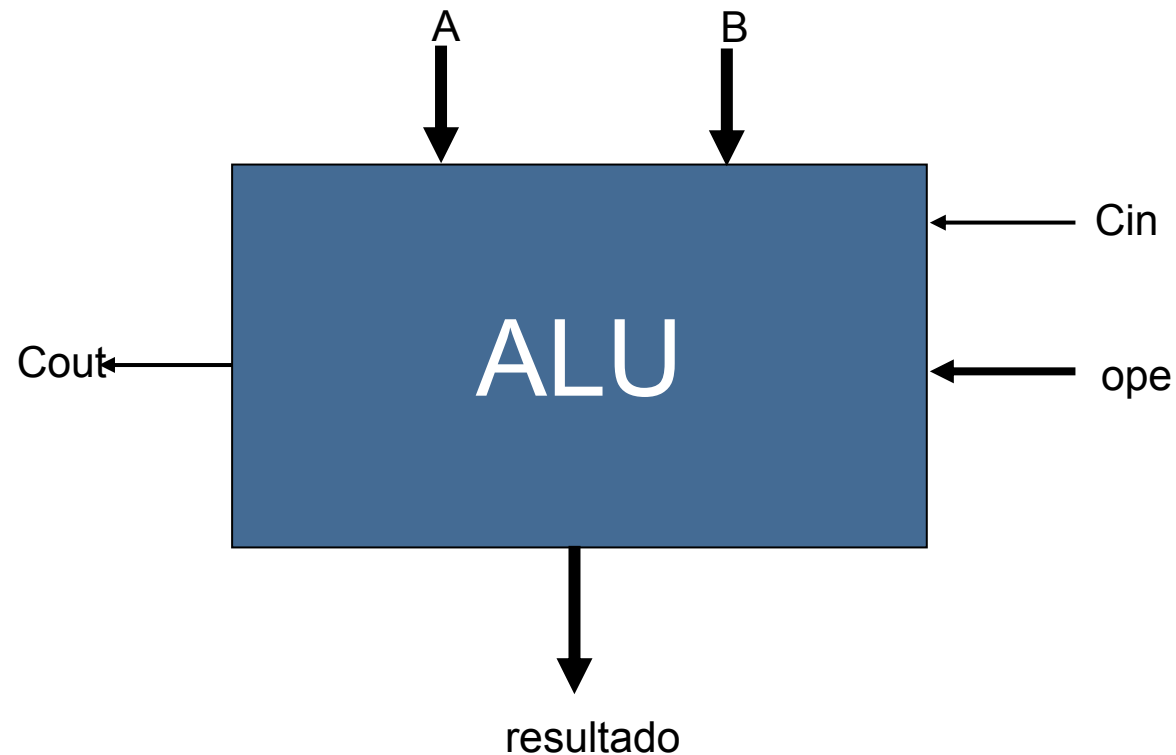
```
if  $p(i)=1$  then  
     $c(i+1) = c(i)$ ;  
else  
     $c(i+1) = g(i)$ ;  
end if;
```





Ejercicio propuesto

1. Modelar en VHDL utilizando COMPONENT una ALU que cumple las siguientes especificaciones:



Ope	Operación	Función
0000	Resultado \leq A	Transparente a A
0001	Resultado \leq A + 1	Incrementa A
0010	Resultado \leq A - 1	Decrementa A
0011	Resultado \leq B	Transparente a B
0100	Resultado \leq B + 1	Incrementa B
0101	Resultado \leq B - 1	Decrementa B
0110	Resultado \leq A - B	Resta
0111	Resultado \leq A + B + Cin	Suma A y B y el Cin
1000	Resultado \leq NOT A	C1(A)
1001	Resultado \leq NOT B	C1(B)
1010	Resultado \leq A AND B	AND
1010	Resultado \leq A OR B	OR
1100	Resultado \leq A NAND B	NAND
1101	Resultado \leq A NOR B	NOR
1110	Resultado \leq A XOR B	XOR
1111	Resultado \leq A XNOR B	XNOR





La multiplicación

- Algoritmo de suma y desplazamiento
- Si multiplicando de n bits y multiplicador de m bits, entonces el producto tendrá una longitud de $n+m$ bits.
- Multiplicación binaria: sencilla ya que hay que multiplicar por 1 o por 0.

Multiplicando				5	3	2
Multiplicador				4	3	1
				<hr/>		
				5	3	2
		1	5	9	6	
	2	1	2	8		
	<hr/>					
Producto	2	2	9	2	9	2



Multiplicación binaria sin signo

Repetir n veces

Si el bit 0 del registro producto=1 entonces

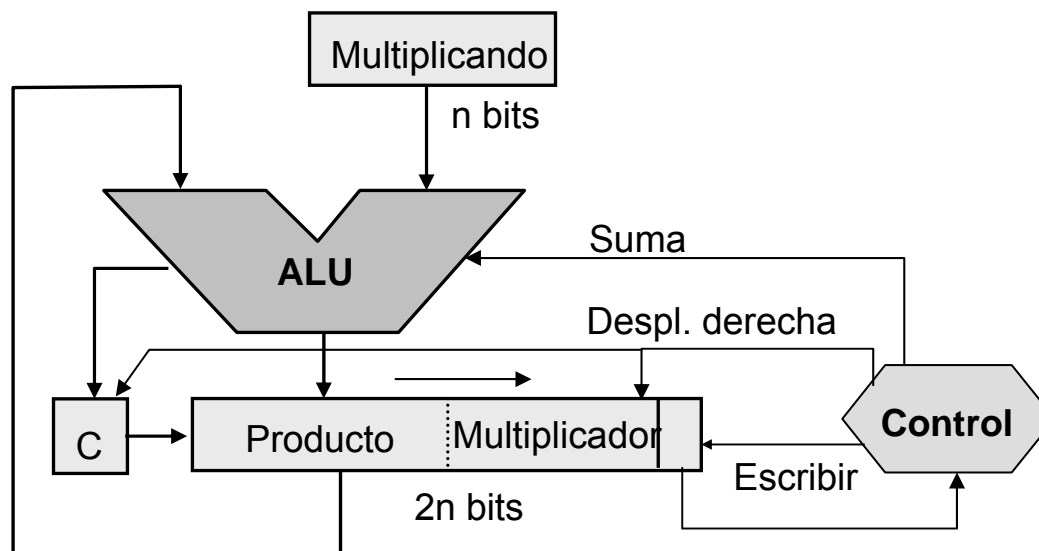
Sumar el multiplicando a la mitad izquierda del producto y colocar el resultado en la mitad izquierda del producto.

Fin entonces

Desplazar 1 bit a la derecha el registro producto

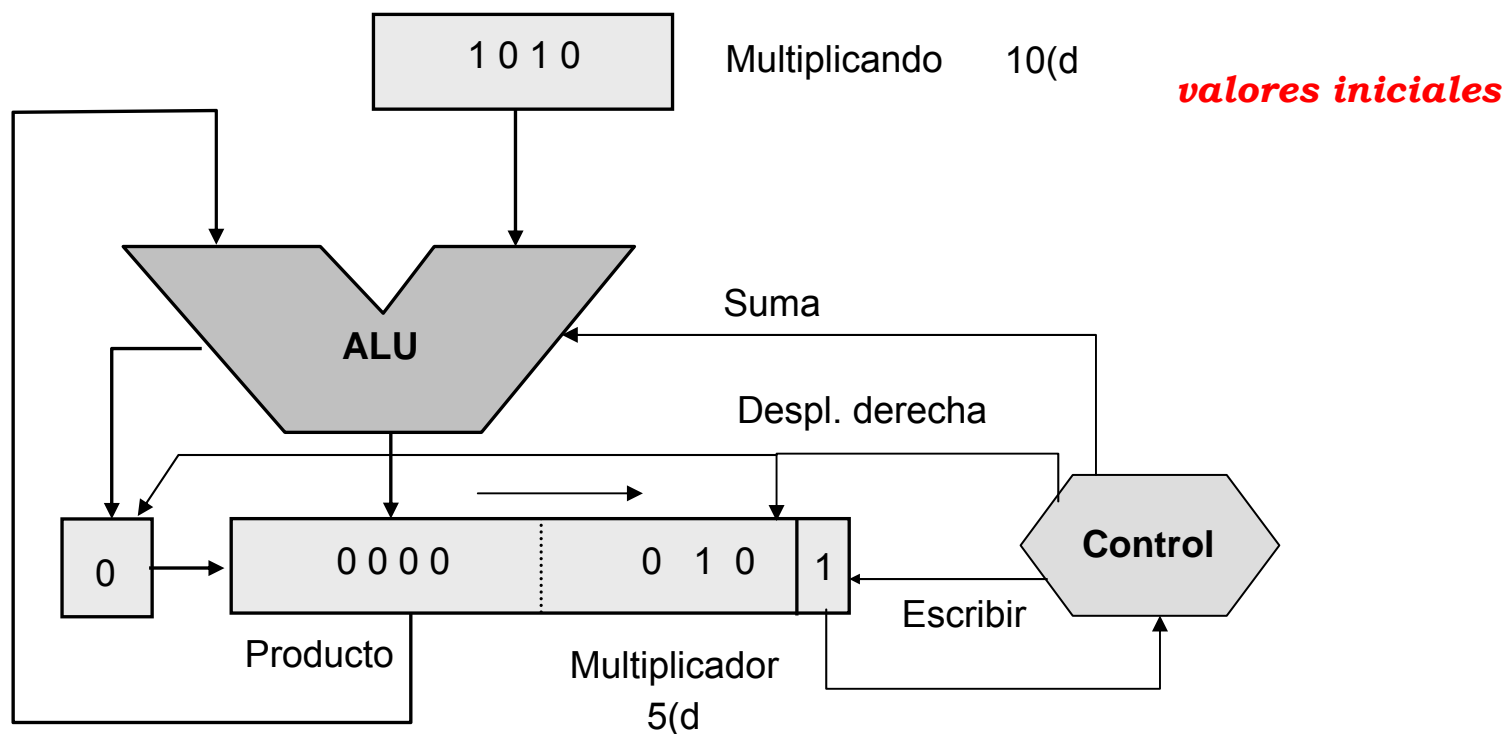
Fin repetir

*Versión
final*



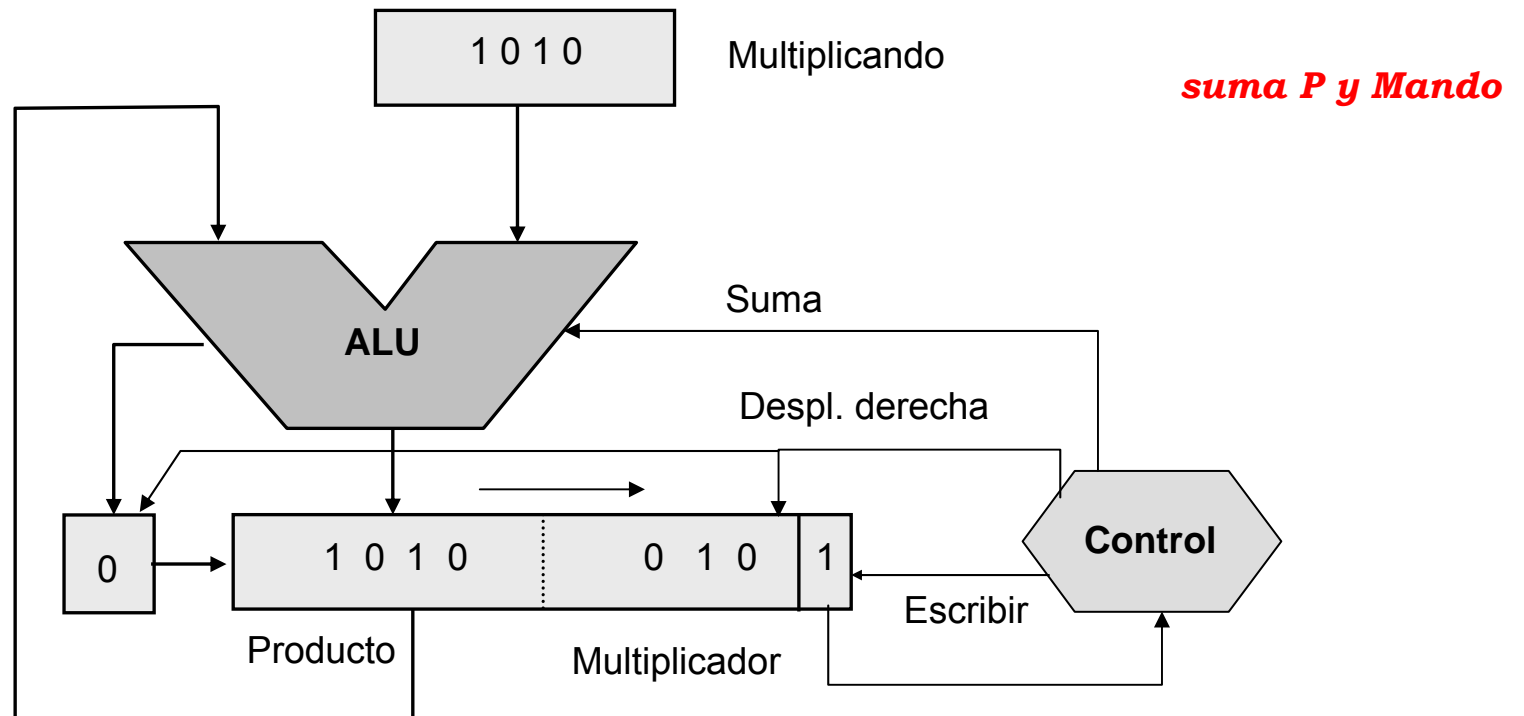


Multiplicación binaria sin signo



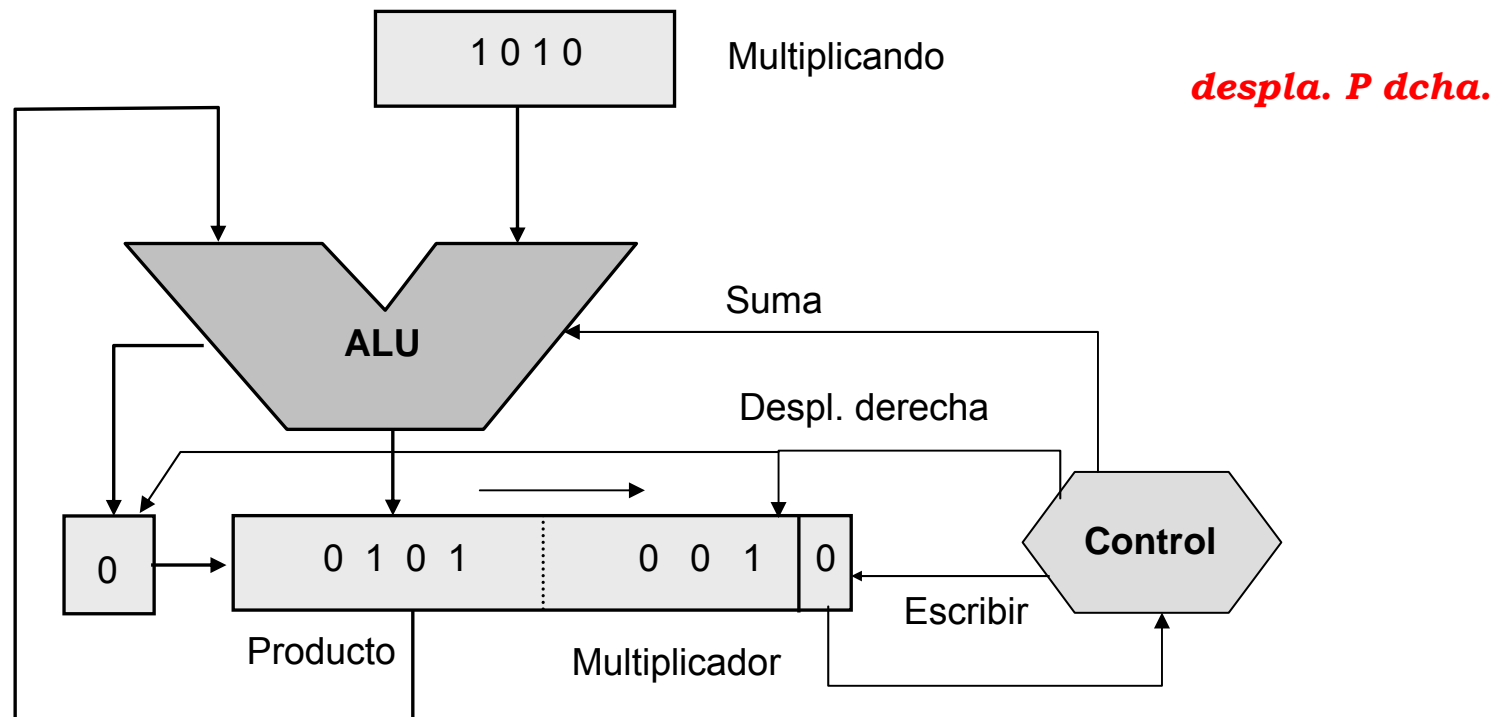


Multiplicación binaria sin signo



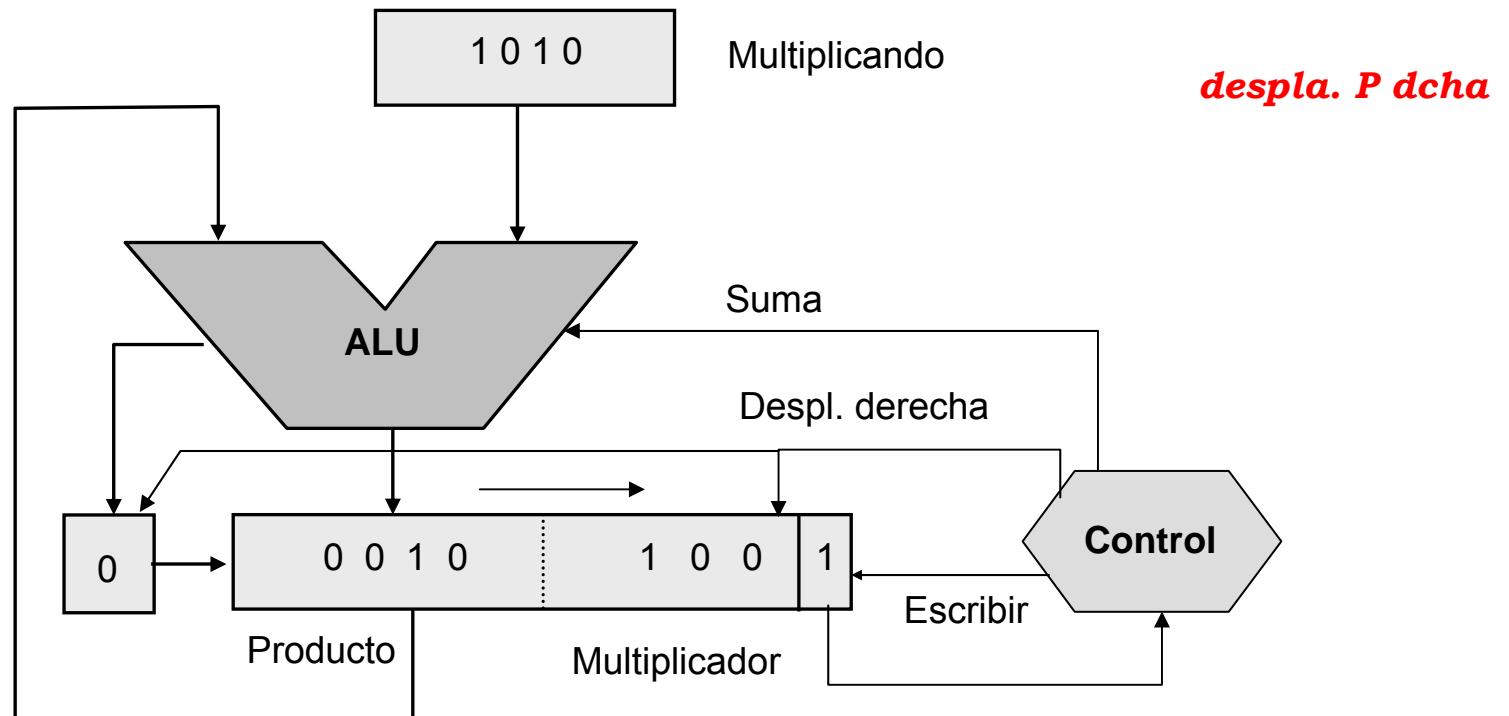


Multiplicación binaria sin signo



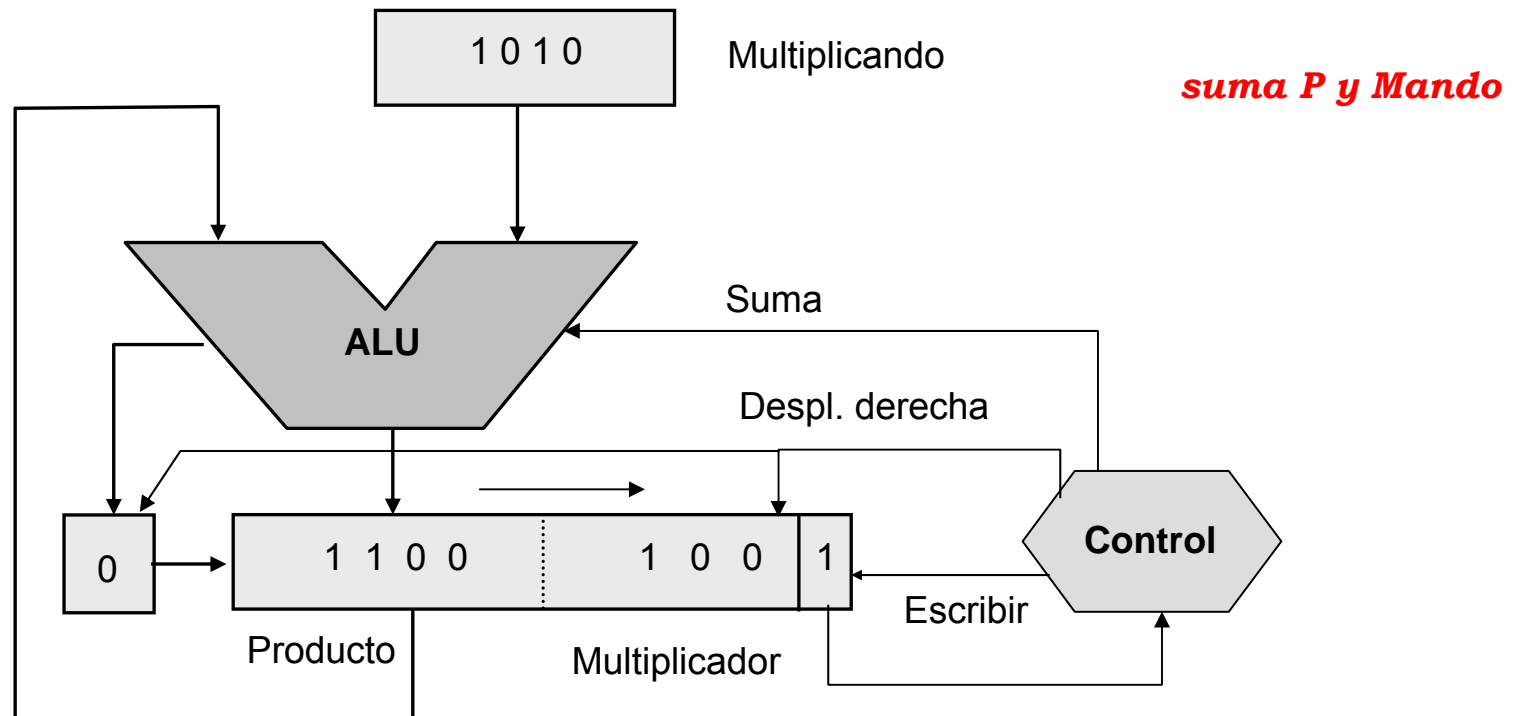


Multiplicación binaria sin signo



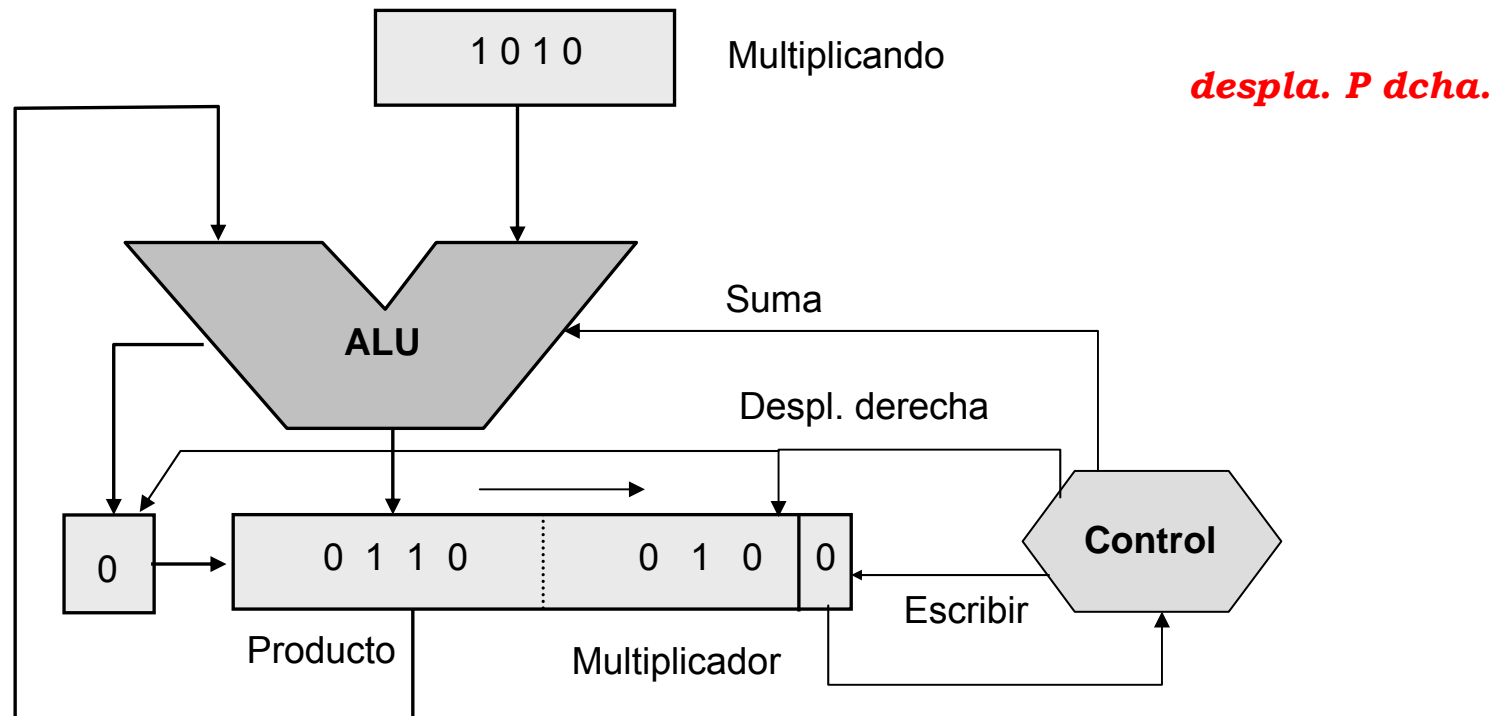


Multiplicación binaria sin signo



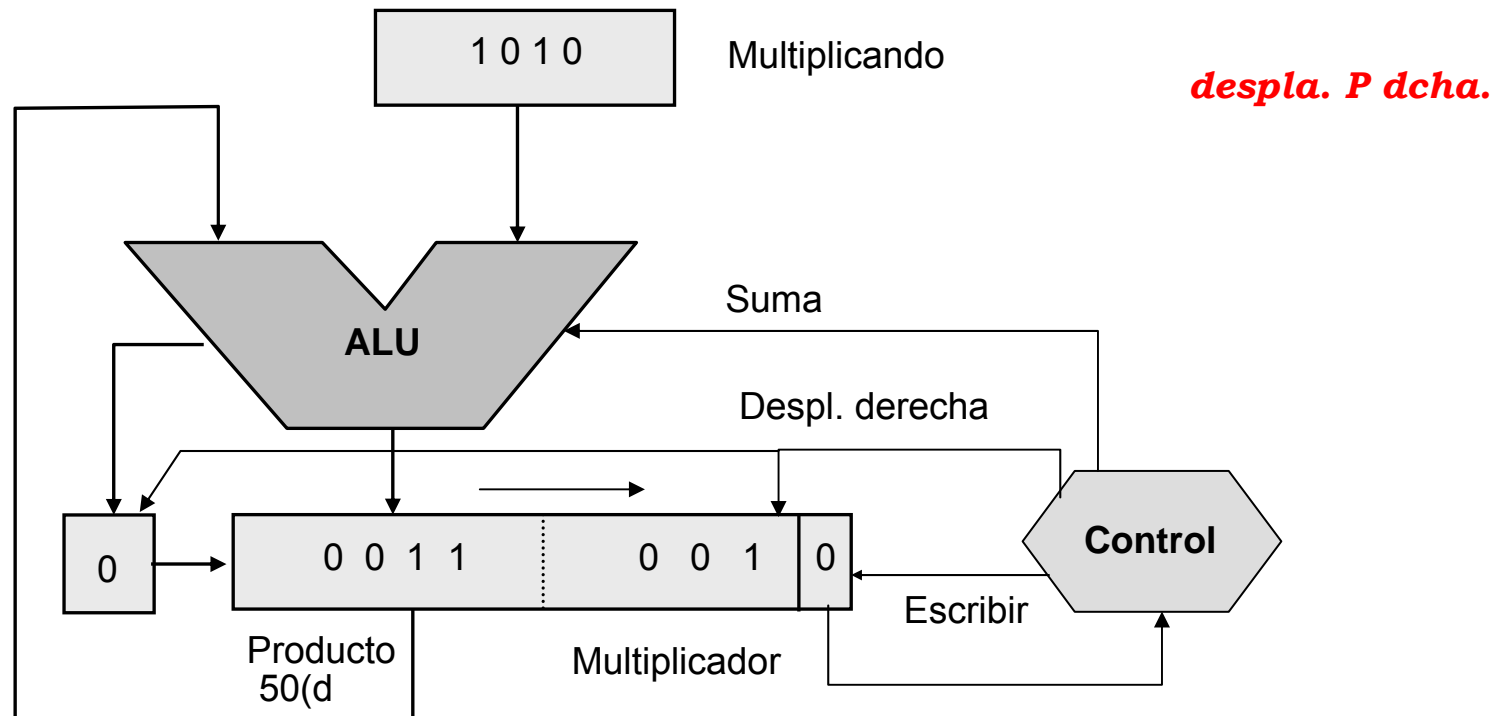


Multiplicación binaria sin signo



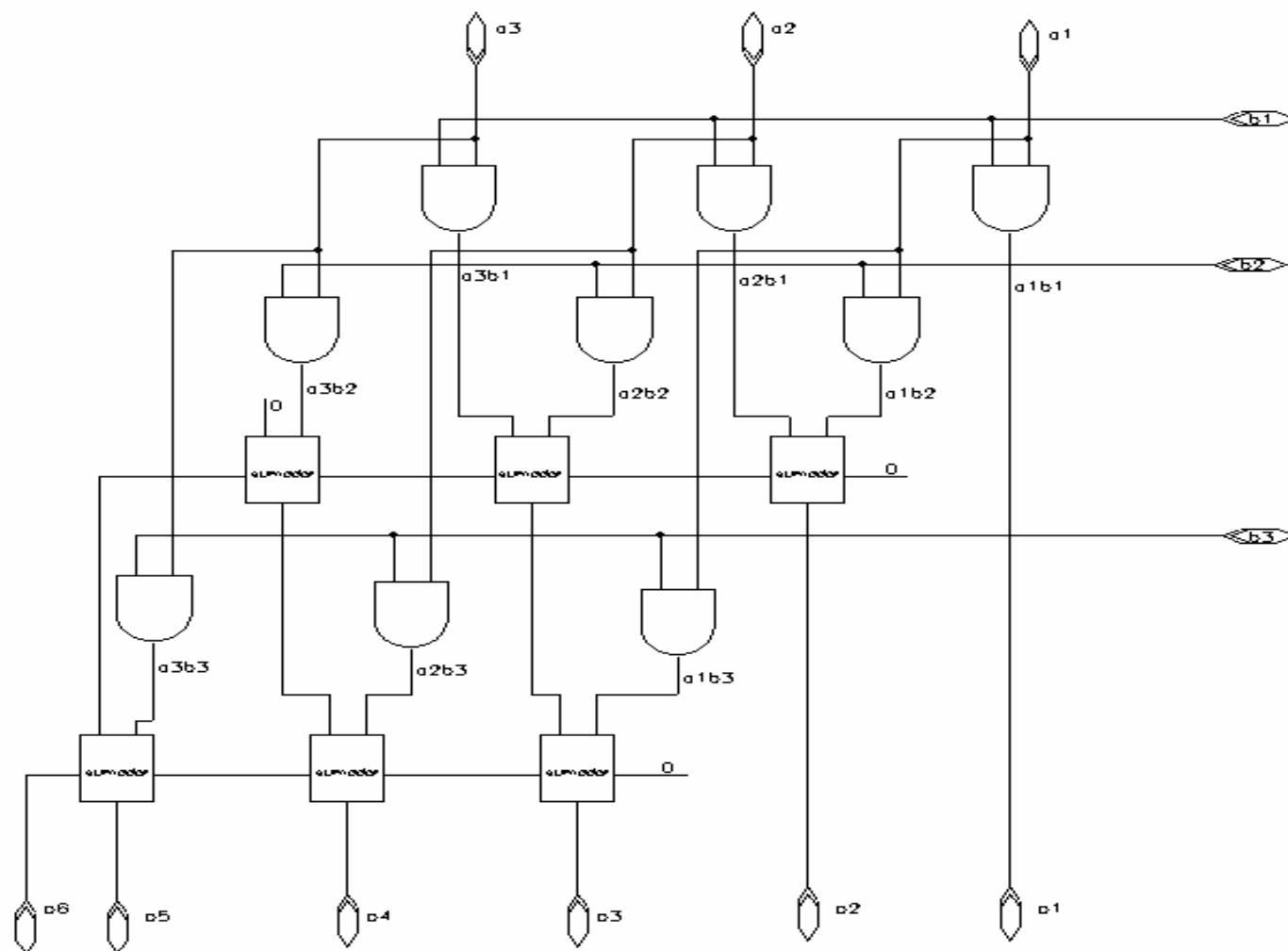


Multiplicación binaria sin signo





Multiplicación rápida





Multiplicación binaria con signo

- Supongamos números expresados en Ca2
- A = 1010 y B = 0011
- Apliquemos algoritmo de sumas y desplazamientos

$$\begin{array}{r} 1010 \\ \times 0011 \\ \hline 1010 \\ 1010 \\ 0000 \\ 0000 \\ \hline 0011110 \end{array}$$

Versión errónea

$$\begin{array}{r} 1010 \\ \times 0011 \\ \hline 11111010 \\ 11111010 \\ 000000 \\ 000000 \\ \hline 11101110 \end{array}$$

Versión correcta



Algoritmo de Booth

- Supongamos Multiplicando = 2 y Multiplicador = 7 (en binario 0010 x 0111)
- Booth expresó $7 = 8 - 1$ y sustituyo el multiplicador por esta descomposición: $0111 = 1000 - 0001 = +100-1$

				0	0	1	0
			x	+1	0	0	-1
1	1	1	1	1	1	1	0
0	0	0	0	0	0		
0	0	0	1	0			
0	0	0	0	1	1	1	0

Multiplicando

Multiplicador según A. Booth

Restamos el multiplicadndo

2 despl. (2 ceros en el multiplicador)

Sumamos el multiplicando



Algoritmo de Booth

Bit actual	Bit a la izquierda	Sustitución
0	0	0 (no hay transición)
0	1	-1 (transición hacia negativo)
1	0	+1 (transición hacia positivo)
1	1	0 (no hay transición)

Ejemplo: Multiplicando = 11101110 y Multiplicador = 01111010

Recodificación del multiplicador según Booth = +1000-1+1-10

								x	1	1	1	0	1	1	1	0
									+1	0	0	0	-1	+1	-1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	
1	1	1	1	1	1	1	1	1	1	0	1	1	1	0		
0	0	0	0	0	0	0	0	0	1	0	0	1	0			
1	1	1	1	0	1	1	1	0	0	0	0	0				
1	1	1	1	0	1	1	1	0	1	1	0	1	1	0	0	



Algoritmo de Booth

Inicialmente $q_{-1}=0$

Repetir n veces

Si $q_0 = 1$ y $q_{-1} = 0$ entonces

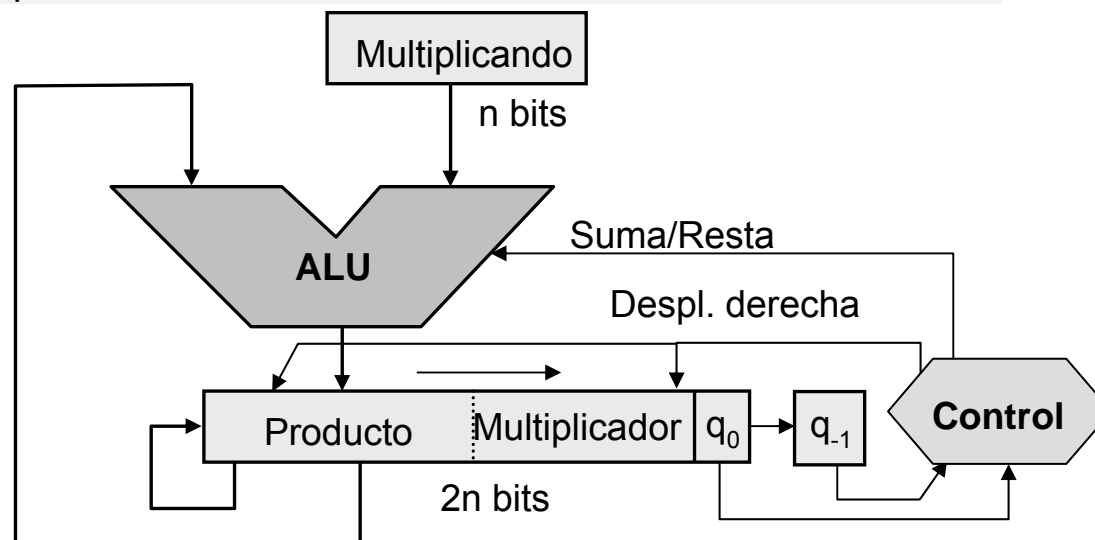
Producto_h = producto_h - Multiplicando

Si $q_0 = 0$ y $q_{-1} = 1$ entonces

Producto_h = Producto_h + Multiplicando

Desplazamiento aritmético a la derecha de Producto y q_{-1}

Fin repetir.





La división

- La división la podemos expresar como:

$$\text{Dividendo} = \text{Cociente} \times \text{Divisor} + \text{Resto}$$

- El resto es más pequeño que el divisor. Hay que reservar el doble de espacio para el dividendo.
- Supondemos operandos positivos.

$$\begin{array}{r} \text{Dividendo} \rightarrow 10010011 \quad \overline{1011} \quad \leftarrow \text{Divisor} \\ 10010 \quad 01101 \quad \leftarrow \text{Cociente} \\ \underline{1011} \\ 001110 \\ \underline{1011} \\ 001111 \\ \underline{1011} \\ 0100 \quad \leftarrow \text{Resto} \end{array}$$



Algoritmo con restauración

Repetir n veces

Desplazar el Dividendo a la izquierda

$\text{Dividendo}_h = \text{Dividendo}_h - \text{Divisor}$

Si $\text{Dividendo}_h < 0$ entonces (no cabe)

$q_0 = 0$

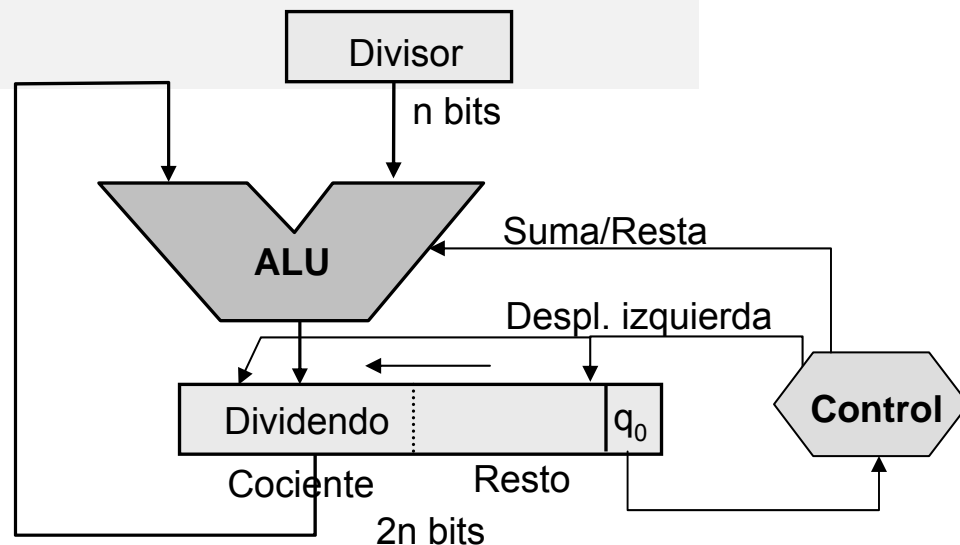
$\text{Dividendo}_h = \text{Dividendo}_h + \text{Divisor}$ (restaurar)

Sino

$q_0 = 1$

Fin Si

Fin Repetir





Algoritmo con restauración

Dividendo	Divisor	Acción	Iteración
0101 0011	0110	Valores iniciales	0
1010 011_	0110	Desplazar un bit a izquierda	1
0100 011_	0110	Restar	1
0100 0111	0110	$\text{Dividendo}_h > 0 \Rightarrow q_0 = 1$	1
1000 111_	0110	Desplazar un bit a izquierda	2
0010 111_	0110	$\text{Dividendo}_h - \text{Divisor}$ (Restar)	2
0010 1111	0110	$\text{Dividendo}_h > 0 \Rightarrow q_0 = 1$	2
0101 111_	0110	Desplazar un bit a izquierda	3
1111 111_	0110	$\text{Dividendo}_h - \text{Divisor}$ (Restar)	3
1111 1110	0110	$\text{Dividendo}_h \leq 0 \Rightarrow q_0 = 0$	3
0101 1110	0110	$\text{Dividendo}_h + \text{Divisor}$ (Restaurar)	3
1011 110_	0110	Desplazar un bit a izquierda	4
0101 110_	0110	$\text{Dividendo}_h - \text{Divisor}$ (Restar)	4
0101 1101	0110	$\text{Dividendo}_h > 0 \Rightarrow q_0 = 1$	4

↑ ↑

Resto Cociente



Algoritmo sin restauración

$\text{Dividendo}_h = \text{Dividendo}_h - \text{Divisor}$

Repetir n veces

Si $\text{Dividendo}_h < 0$ entonces

Desplazar el Dividendo a la izquierda

$\text{Dividendo}_h = \text{Dividendo}_h + \text{Divisor}$

Sino

Desplazar el Dividendo a la izquierda

$\text{Dividendo}_h = \text{Dividendo}_h - \text{Divisor}$

Fin Si

Si $\text{Dividendo}_h < 0$ entonces

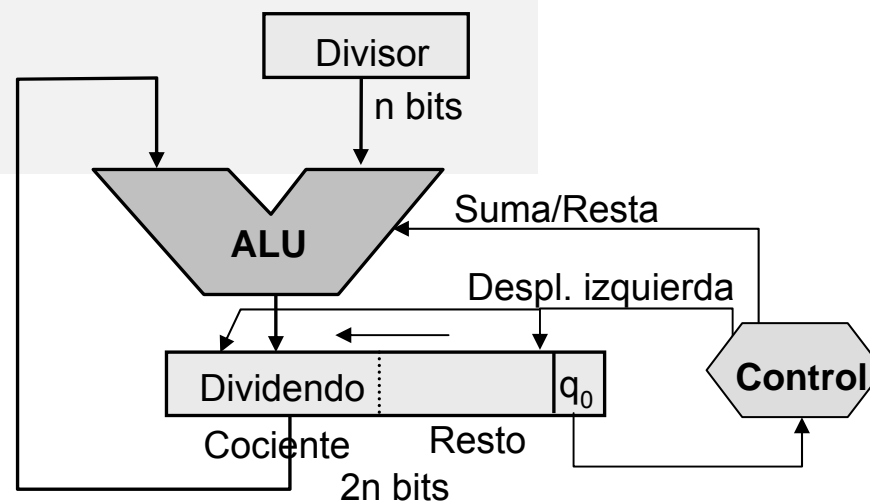
$q_0 = 0$

Sino

$q_0 = 1$

Fin Si

Fin Repetir





Algoritmo sin restauración

Dividendo	Divisor	Acción	Iteración
0000 0111	0010	Valores iniciales	0
1110 0111	0010	$\text{Dividendo}_h - \text{Divisor}$	0
1100 111_	0010	$\text{Dividendo}_h < 0 \Rightarrow$ Desplazar Izda	1
1110 111_	0010	$\text{Dividendo}_h + \text{Divisor}$	1
1110 1110	0010	$\text{Dividendo}_h < 0 \Rightarrow q_0 = 0$	1
1101 110_	0010	$\text{Dividendo}_h < 0 \Rightarrow$ Desplazar Izda	2
1111 110_	0010	$\text{Dividendo}_h + \text{Divisor}$	2
1111 1100	0010	$\text{Dividendo}_h < 0 \Rightarrow q_0 = 0$	2
1111 100_	0010	$\text{Dividendo}_h < 0 \Rightarrow$ Desplazar Izda	3
0001 100_	0010	$\text{Dividendo}_h + \text{Divisor}$	3
0001 1001	0010	$\text{Dividendo}_h \geq 0 \Rightarrow q_0 = 1$	3
0011 001_	0010	$\text{Dividendo}_h > 0 \Rightarrow$ Desplazar Izda	4
0001 001_	0010	$\text{Dividendo}_h - \text{Divisor}$	4
0001 0011	0010	$\text{Dividendo}_h > 0 \Rightarrow q_0 = 1$	4

↑ ↑

Resto Cociente



Conclusiones

● Sumadores

- Problemática temporal de los Sumadores con Propagación de Acarreo (CPA), especialmente si n elevado.
- Los Sumadores con anticipación de acarreo (CLA) mejoran el tiempo de respuesta de los sumadores.

● Multiplicación

- Problemática de la multiplicación de números con signo.
- El algoritmo de Booth permite multiplicar números en $Ca2$ y en algunos casos reduce el número de operaciones si aparecen cadenas de 1's o 0's en el multiplicador.

● La División

- Algoritmo para la división con restauración para números positivos. Si números negativos, entonces tratamiento previo del signo, y en función de éste se obtiene el signo del resultado.



Coma flotante

- Representación para números fraccionarios
 - Coma fija 1234,567
 - Logarítmica $\log 123,456 = 2,0915122$
 - Coma flotante $1,234566 \times 10^3$
 - Otras
- Ventajas de estandarizar una representación determinada
 - Posibilidad de disponer de bibliotecas de rutinas aritméticas
 - Técnicas de implementación en hardware de alto rendimiento
 - Construcción de aceleradores aritméticos estándar, etc.
- En la actualidad la industria de los computadores está convergiendo hacia el formato del estándar 754-1985 del IEEE.



Estándar IEEE 754 para coma flotante

Formatos

- Simple precisión (32 bits)

1 bit	8 bits	23 bits
signo	exponente	mantisa

- Doble precisión (64 bits)

1 bit	11 bits	52 bits
signo	exponente	mantisa



Estándar IEEE 754 para coma flotante

- Base del exponente 2
- Exponente representado en exceso $2^{q-1}-1$
 - Exceso a 127 en simple precisión
 - Exceso a 1023 en doble precisión
- Mantisa en valor absoluto; fraccionaria y normalizada con un uno implícito a la izquierda de la coma decimal.
 - Mantisa de la forma 1,XXXXXX
 - El primer uno nunca estará representado
 - Valores posibles entre 1,00000..... y 1,11111.....
- S es el signo de la mantisa
- Números
 - $(-1)^S \times 1,M \times 2^{E-127}$ simple precisión
 - $(-1)^S \times 1,M \times 2^{E-1023}$ doble precisión



Estándar IEEE 754 para coma flotante

Casos especiales

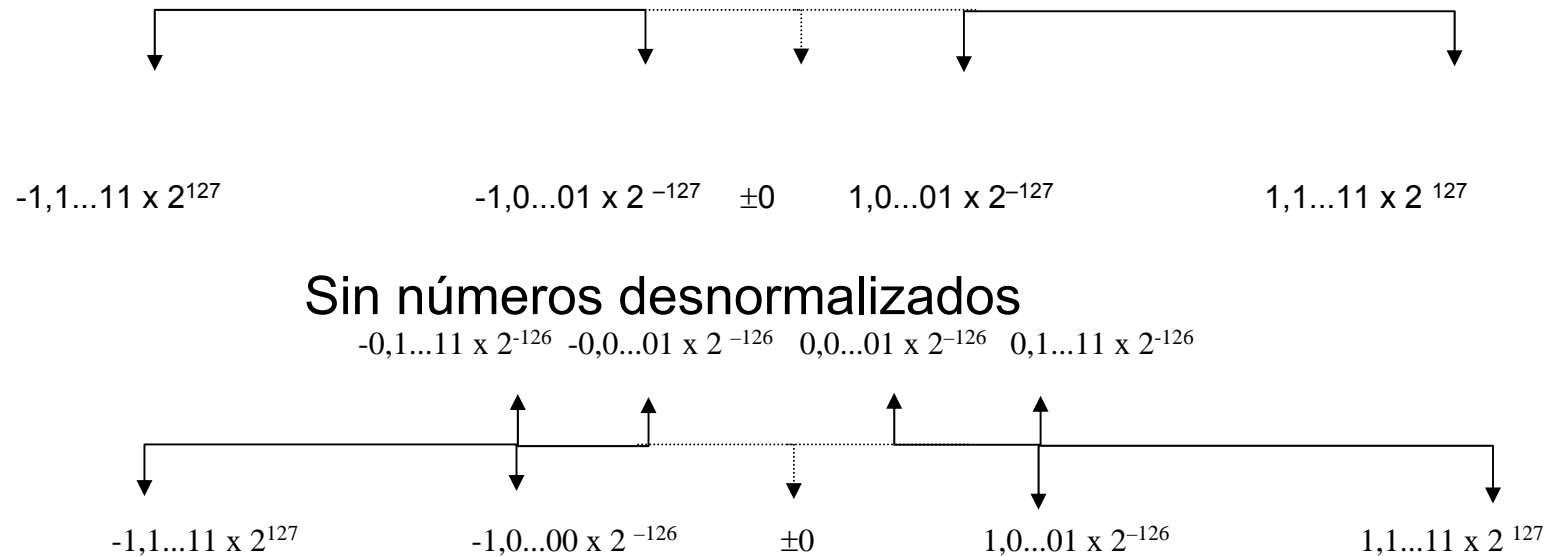
E	M	Valores
$2^{q-1}-1$	$\neq 0$	NaN (no un Número)
$2^{q-1}-1$	0	$+\infty$ y $-\infty$ según el signo de S
0	0	Cero
0	$\neq 0$	Números desnormalizados

- NaN resultado de operaciones tales como $0/0$,
- El valor cero tiene dos representaciones $+0$ y -0 .

Estándar IEEE 754 para coma flotante

Formato desnormalizado

- $0, M \times 2^{-126}$ simple precisión
- $0, M \times 2^{-1022}$ doble precisión



Con números desnormalizados



Operaciones en coma flotante

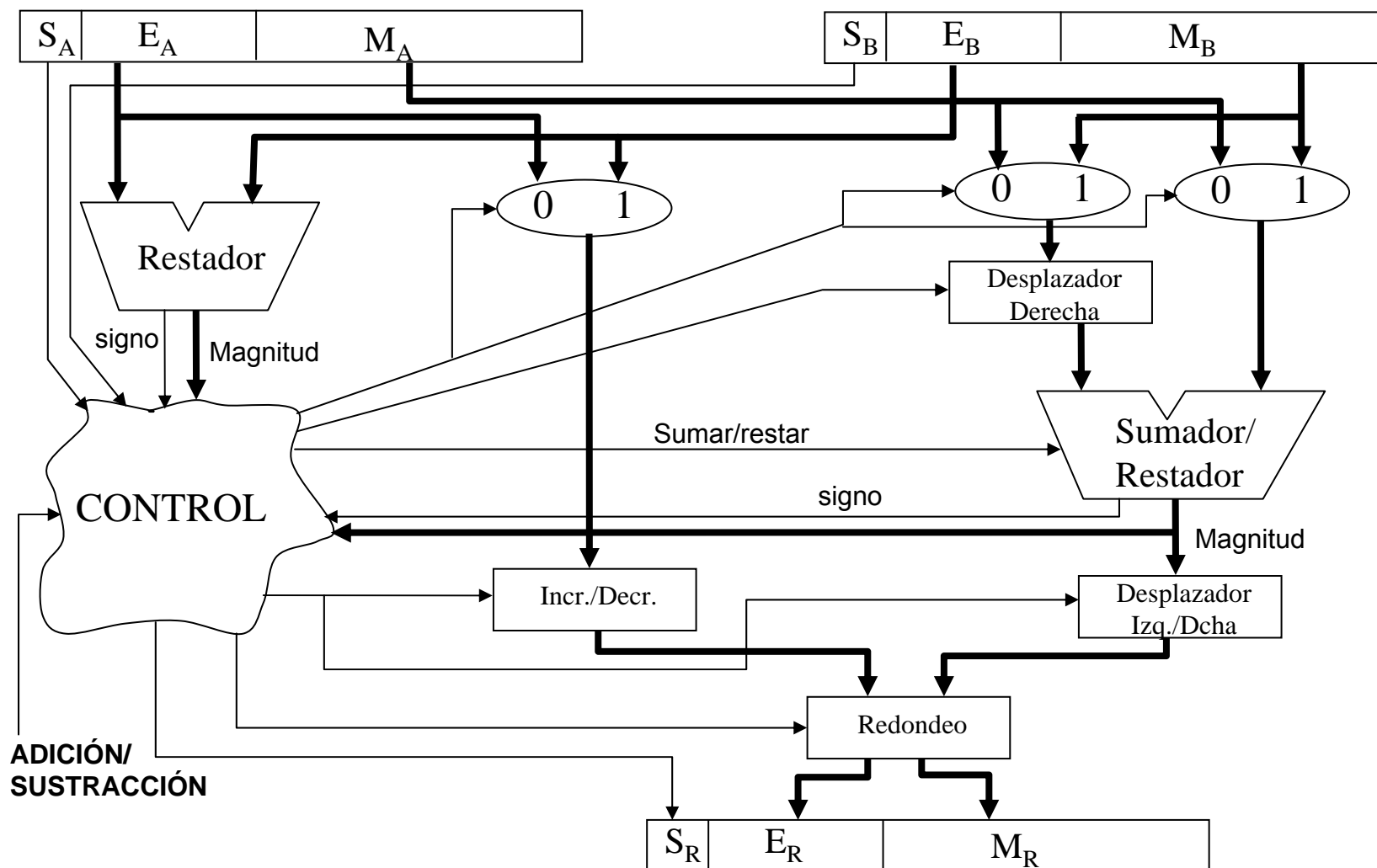
● Operaciones aditivas

■ Reglas de Suma/Resta

1. Seleccionar el número de menor exponente y desplazar su mantisa hacia la derecha tantas posiciones como la diferencia de los exponentes en valor absoluto.
2. Igualar el exponente del resultado al exponente mayor.
3. Operar las mantisas (según operación seleccionada y signos de ambos números) y obtener el resultado en signo y valor absoluto.
4. Normalizar el resultado y redondear la mantisa al número de bits apropiado.



Circuito Sumador/Restador





Multiplicación y división

Reglas de Multiplicación

1. Sumar los exponentes y restar el exceso para obtener el exponente del resultado
2. Multiplicar las mantisas para determinar la mantisa del resultado
3. Procesar los signos
4. Normaliza y redondear si es necesario



Multiplicación y división

- Reglas de División

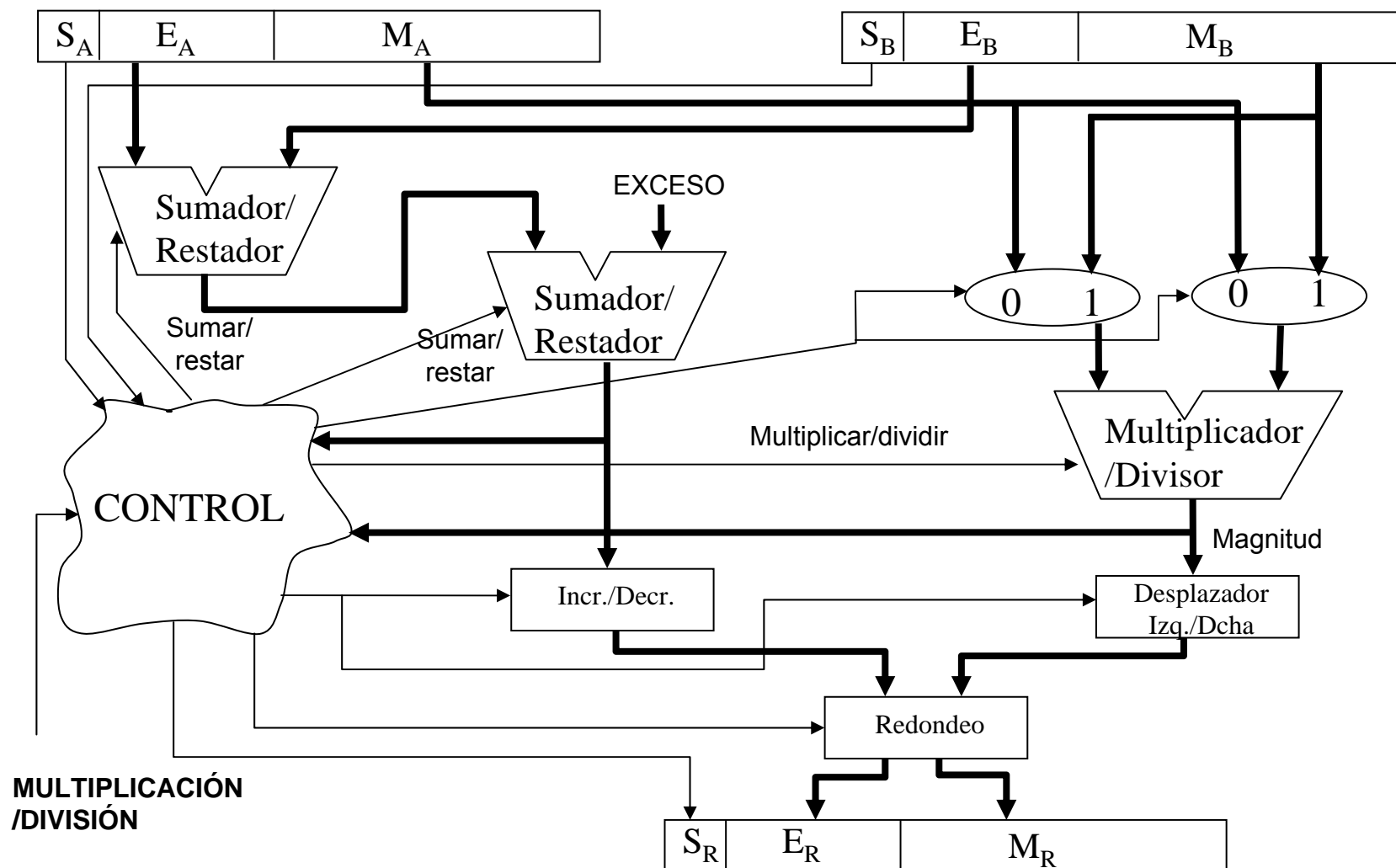
1. Restar los exponentes y sumar el exceso para obtener el exponente resultado
2. Dividir las mantisas para determinar la mantisa del resultado.
3. Procesar los signos.
4. Normalizar y redondear si es necesario.

- Procesamiento de los signos

S_A	S_B	S_R
0	0	0
0	1	1
1	0	1
1	1	1
$S_R = S_A \oplus S_B$		



Circuito Multiplicador /Divisor





Redondeo

- Las técnicas de redondeo consisten en limitar el número de bits al disponible en el sistema de representación utilizado.
- Dada una cantidad **C**, y un **sistema de representación** que permite representar los valores **V_0, V_1, \dots, V_r** .
- El redondeo consiste en asignar a **C** una representación **R** que se le aproxime.
Si $V_{i-1} < C < V_i$ el redondeo consiste en asignar V_{i-1} o V_i como representación R de la cantidad C
- El error absoluto se define como: $\varepsilon = |R - C|$
- La resolución se define como: $\Delta = |V_i - V_{i-1}|$
- Técnicas de redondeo
 - Truncamiento
 - Redondeo propiamente dicho
 - Bit menos significativo forzado a “uno”



Truncamiento

- Elimina los bits a la derecha que no caben en la representación.
 - Es facil de implementar.
 - El error del resultado es siempre por defecto.
 - El error puede crecer rápidamente



Redondeo al más próximo

- Toma el valor más próximo al que se quiere representar

Si $|V_{i-1} - C| < |V_i - C|$ entonces

$$R \equiv V_{i-1}$$

si no

$$R \equiv V_i$$

Ejemplo: representación con 8 bits de punto implícito

$$\begin{array}{lll} C = 0,01100000\overset{\cdot}{0}1 & \equiv & 0,375976563 \\ V_{i-1} = 0,01100000 & \equiv & 0,375 \\ V_i = 0,01100001 & \equiv & 0,37890625 \\ |V_{i-1} - C| = 0,000976563 & & \\ |V_i - C| = 0,00390625 & \longrightarrow & R = 0,01100000 \end{array}$$



Bit menos significativo forzado a “uno”

- Consiste en truncar y forzar el bit menos significativo a “uno”
 - Es muy rápido, tanto como el truncamiento
 - Sus errores son tanto por defecto como por exceso.