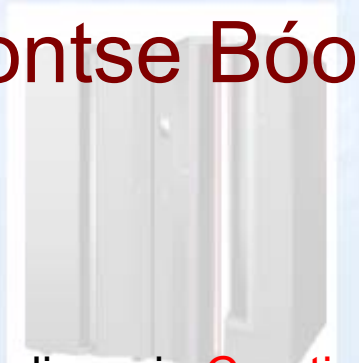


# Aritmética del computador

Montse Bóo Cepeda





# Estructura del curso

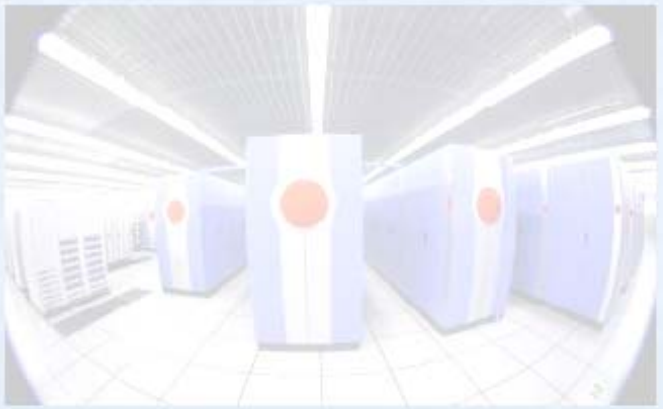
1. Evolución y caracterización de los computadores.
2. Arquitectura del MIPS: Introducción.
3. Tipo de datos.
4. El repertorio de instrucciones.
5. Aritmética del computador.
6. El camino de datos.
7. Sección de control.
8. El camino de datos multiciclo.
9. Sección de control multiciclo.
10. Entrada/Salida (I/O).

# Esquema de contenidos

1. Punto fijo
   - Operaciones lógicas
   - Suma y resta
   - Multiplicación
   - Multiplicación con signo
2. Punto flotante
   - Suma
   - Multiplicación

# Punto fijo



# Unidad aritmética y lógica: ALU (Arithmetic Logic Unit)

ALU: Dispositivo que realiza las operaciones aritméticas (suma, resta,…) y lógicas (AND, OR,…).

Construcción de una ALU usando puertas lógicas:

- AND
- OR
- Inversor
- Multiplexor (mux)

Objetivo: Diseño modular = Divide y vencerás

- Módulo: ALU de 1 bit.
- ALU de 32 bits: 32 módulos de 1 bit.

Diseño de la ALU de 1 bit:

- Implementación de todas las operaciones en paralelo.
- Selección de la operación con un multiplexor.
- Ventaja: FÁCIL EXTENSIÓN A MÁS OPERACIONES.

# ALU: Operaciones en punto fijo

1. Operaciones lógicas
2. Suma y resta
3. Multiplicación
4. Multiplicación con signo
5. División

Las estudiaremos en este curso

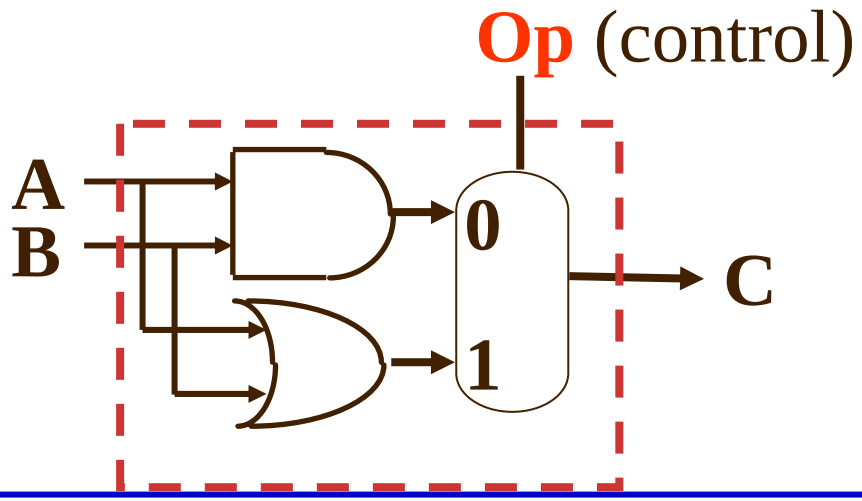
# ALU de 1 bit: AND y OR

Proyección directa en componentes hardware:

- Operación AND: Puerta AND
- Operación OR: Puerta OR

Diseño de una ALU de 1 bit:

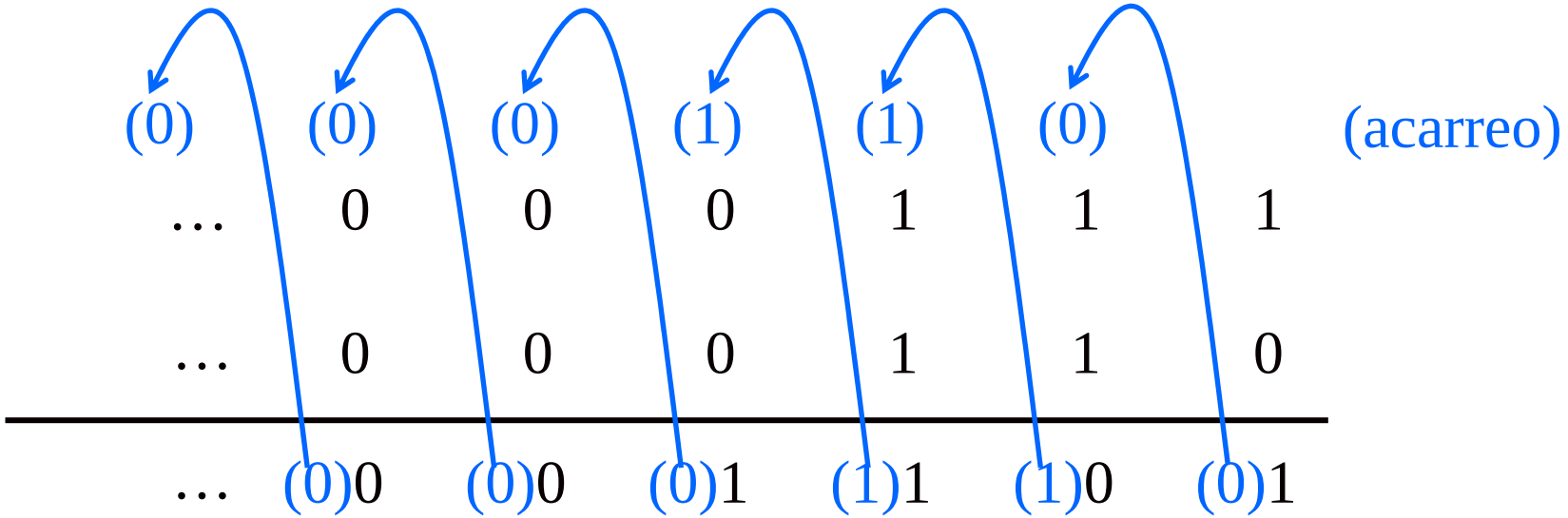
- Puertas AND y OR
- Multiplexor de selección




| Op | C |
|---|---|
| 0 | A and B |
| 1 | A or B |

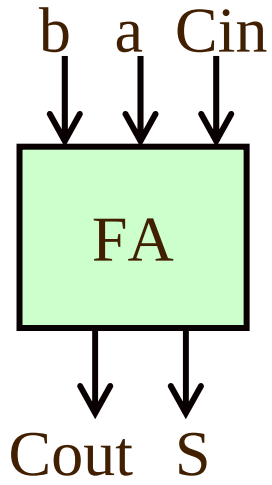
# Suma y resta

- Suma: Los dígitos se suman bit a bit de derecha a izquierda y los acarreos se pasan de un bit al siguiente.
- Resta: Se niega el sustraendo antes de sumar.
- Ejemplo: Sumar $(6)_{diez}$ más $(7)_{diez}$



# Full Adder (FA): Sumador completo de 1 bit

Un full adder tiene 3 bits de entrada
y genera 2 bits de salida: suma y acarreo.




| 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 |

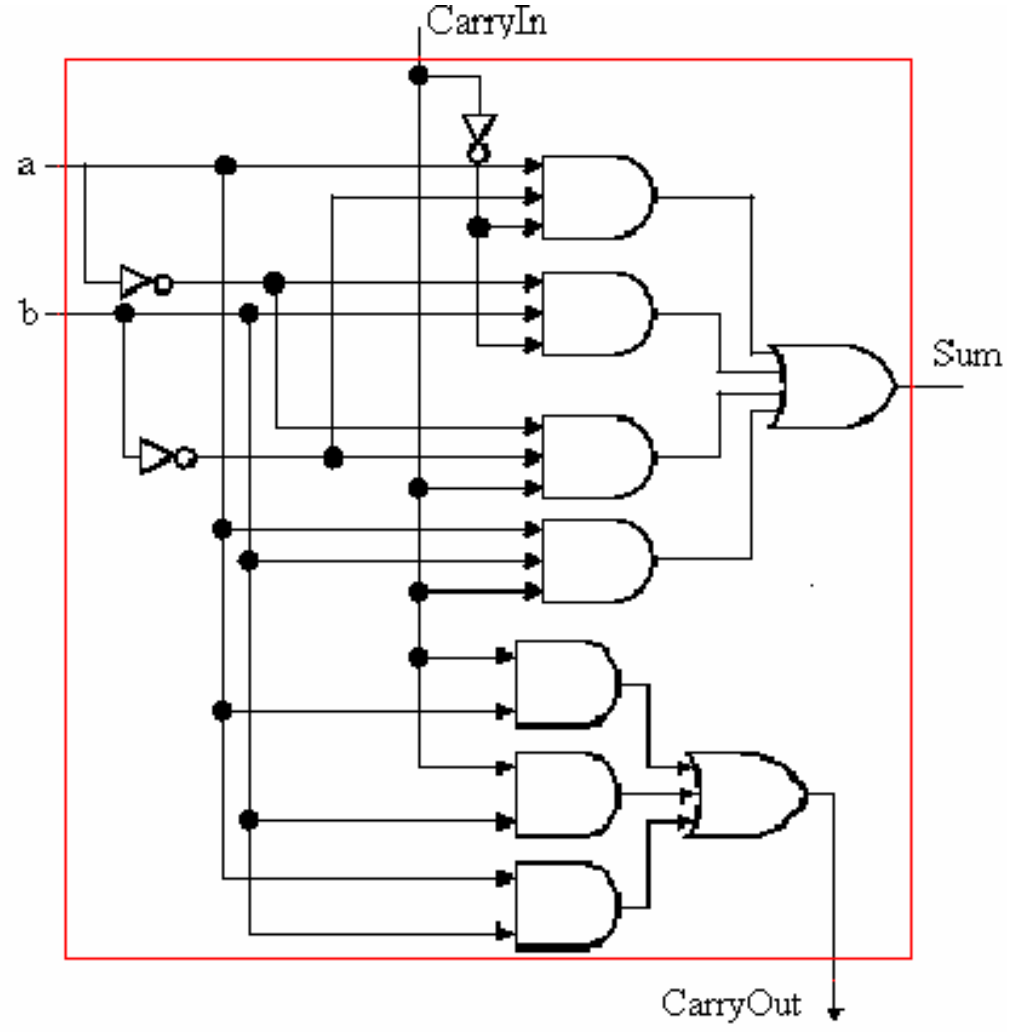
# Full Adder (FA): Sumador completo de 1 bit

| 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 |

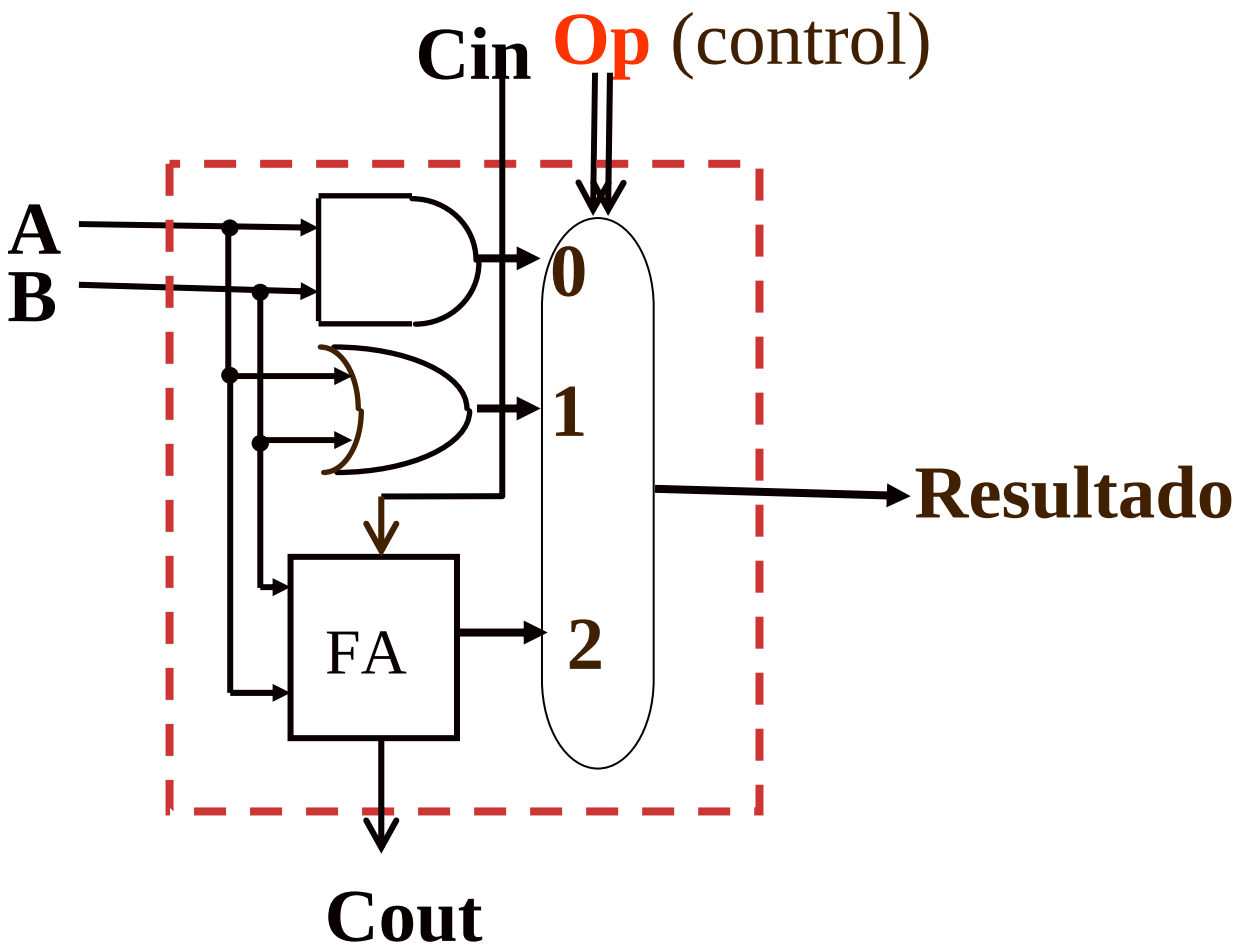
**Cout = (A'*B*Cin) + (A*B'*Cin) + (A*B*Cin') + (A*B*Cin) = (B*Cin) + (A*Cin) + (A*B)**

**S = (A'*B'*Cin) + (A'*B*Cin') + (A*B'*Cin') + (A*B*Cin)**

# Full Adder (FA): Sumador completo de 1 bit



# ALU de 1 bit: AND, OR, suma

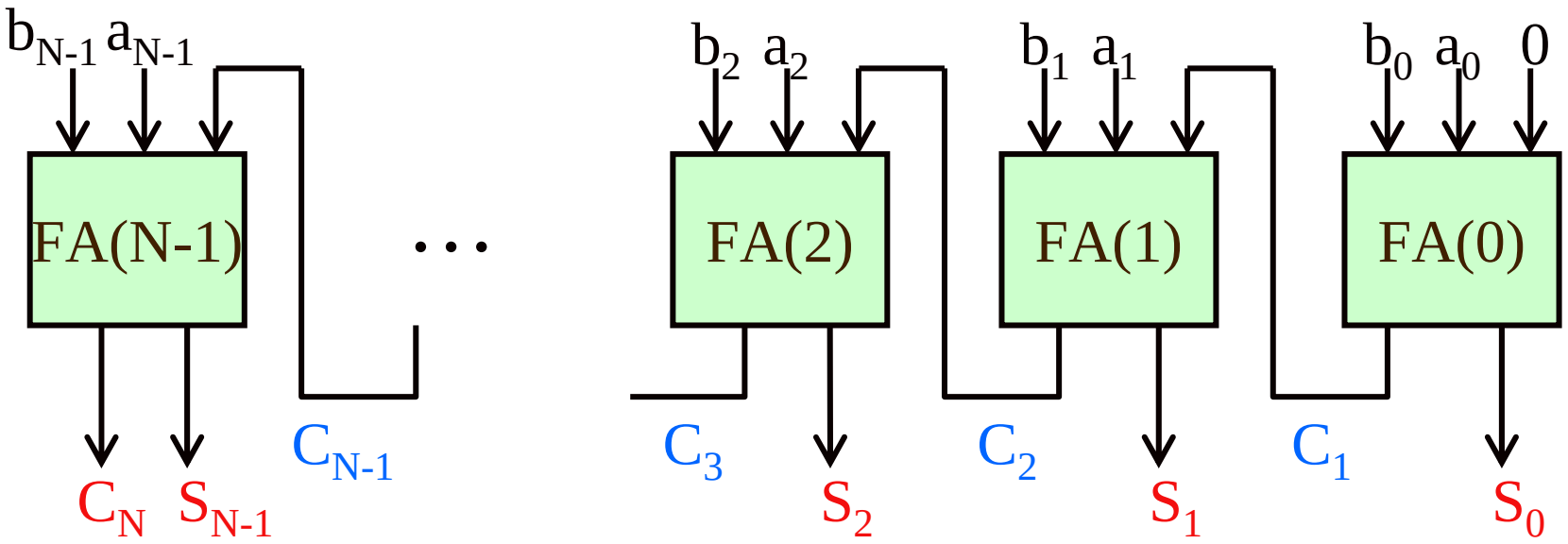


# Carry Ripple Adder: Sumador de acarreo enlazado

Sumador de N bits : N full adders

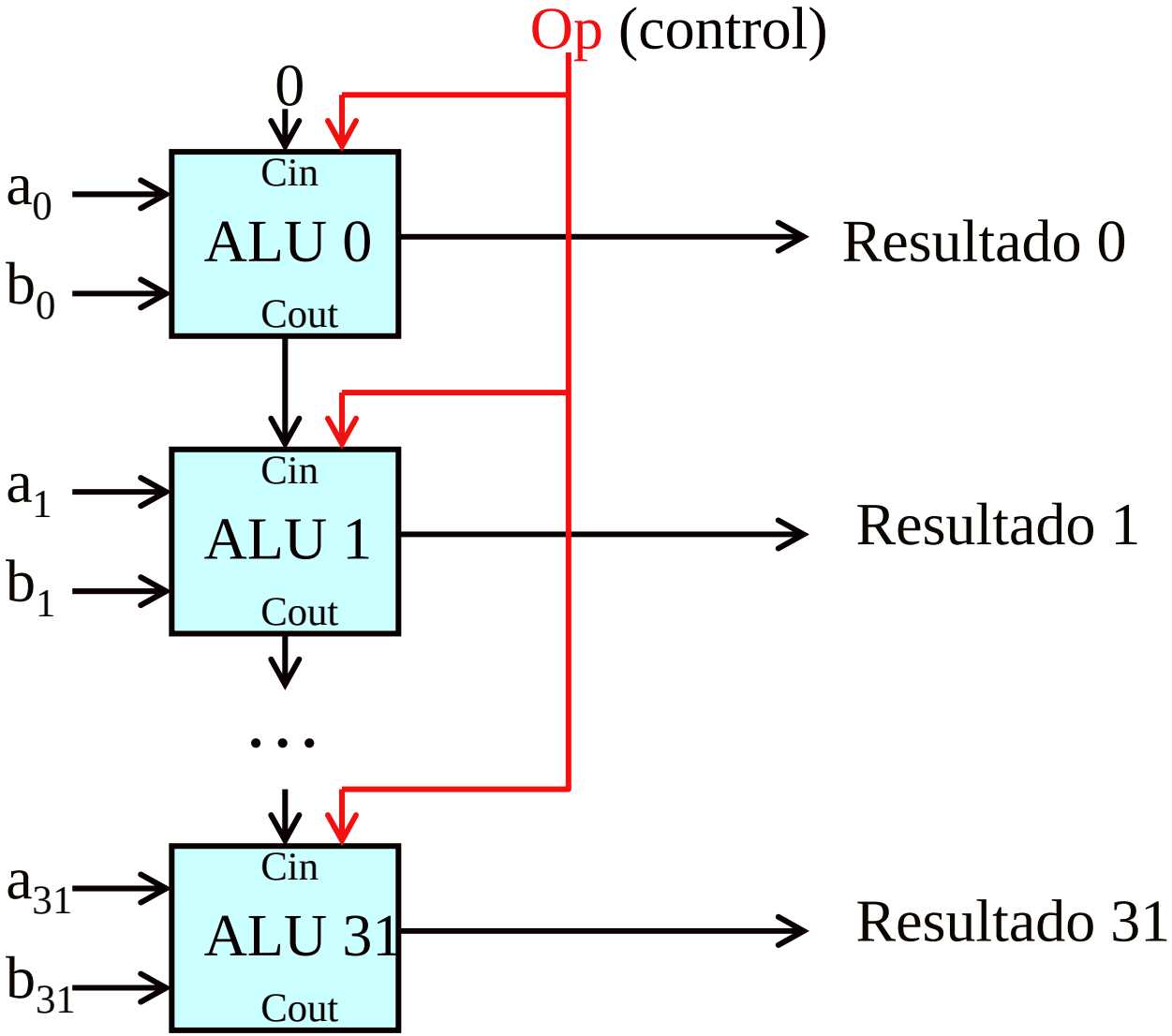
Entradas: $a = a_{N-1}\ a_{N-2} \ldots a_2\ a_1\ a_0$

$b = b_{N-1}\ b_{N-2} \ldots b_2\ b_1\ b_0$



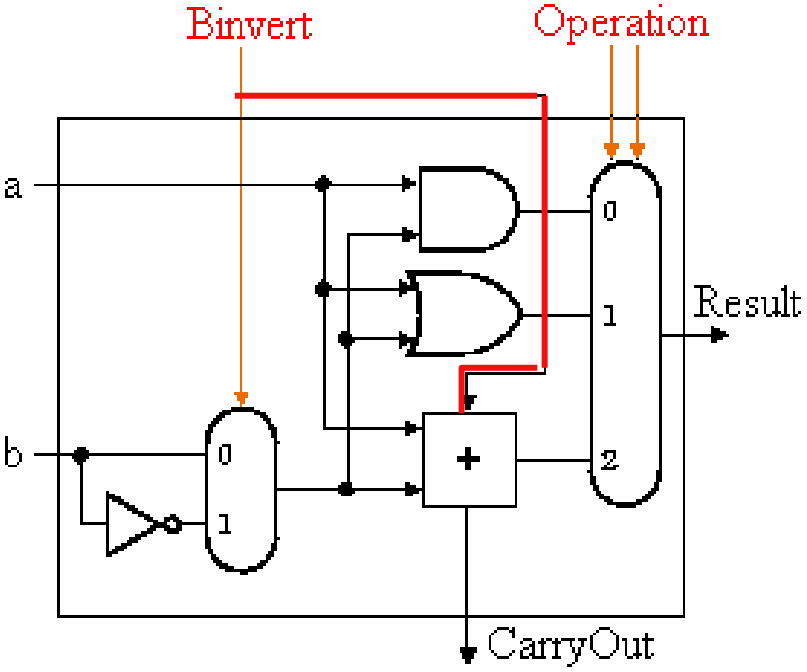

Resultado: $C_N\ S_{N-1}\ S_{N-2} \ldots S_2\ S_1\ S_0$

# ALU de 32 bits: AND, OR, suma

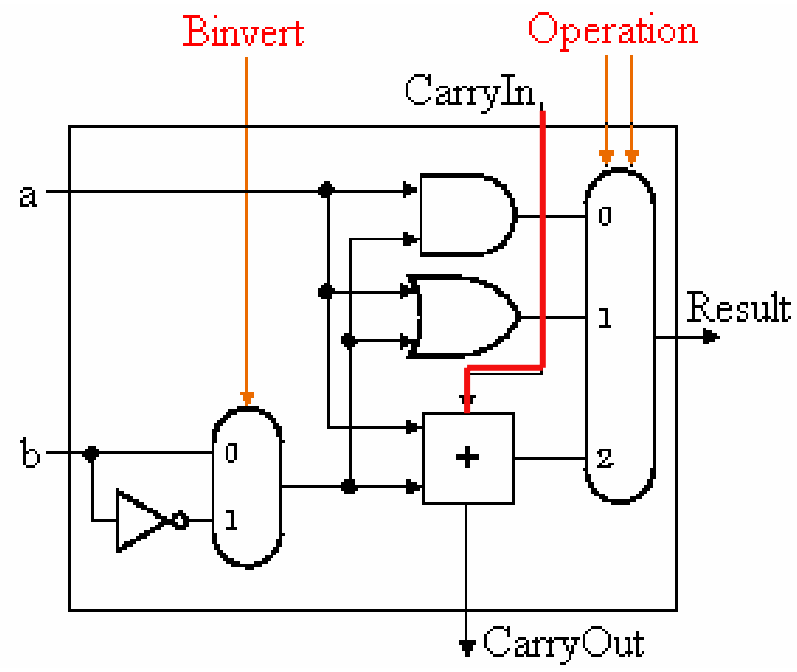


# ALU de 1 bit: AND, OR, suma, resta

Resta: Reutilización del hardware para la suma → Se niega el sustraendo antes de sumar.

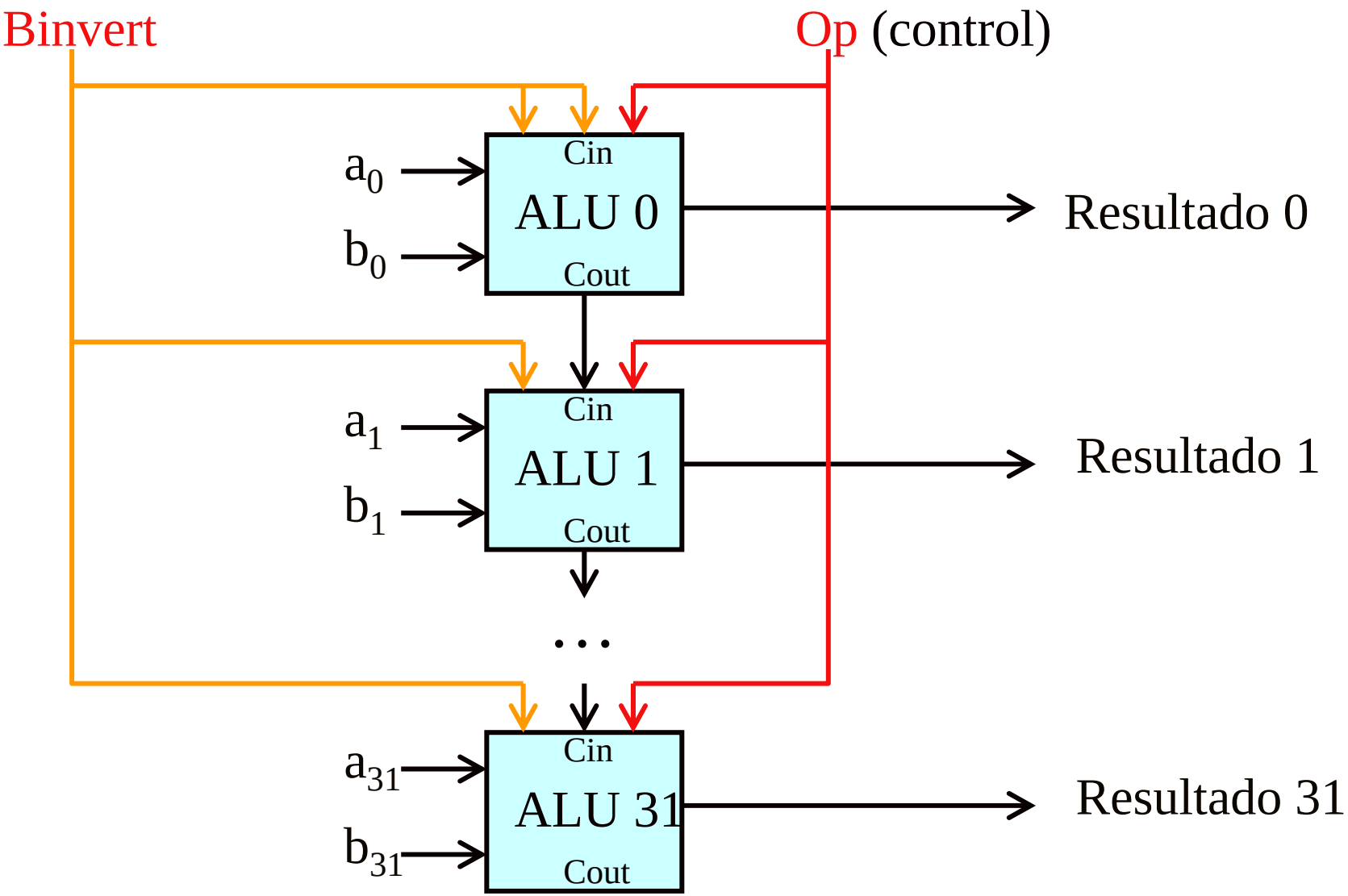



BIT MENOS SIGNIFICATIVO




BITS RESTANTES

# ALU de 32 bits: AND, OR, suma,resta



# Desbordamiento

- Desbordamiento (overflow): El resultado obtenido supera el rango asociado al número de bits utilizado (el resultado no puede codificarse con los bits disponibles)

| TAMAÑO | SIN SIGNO | CON SIGNO (C-2) |
|---|---|---|
| N bits | $[0, 2^N-1]$ | $[-2^{N-1}, 2^{N-1}-1]$ |
| Byte (8 bits) | [0, 255] | [-128, 127] |
| 16 bits | [0, 65535] | [-32768, 32767] |
| 32 bits | [0, 4294967295] | [-2147483648, 2147483647] |

# Desbordamiento

- Condiciones de desbordamiento para la suma y la resta

| Operación | Operando A | Operando B | Resultado |
|---|---|---|---|
| A+B | ≥ 0 | ≥ 0 | <0 |
| A+B | <0 | <0 | ≥ 0 |
| A-B | ≥ 0 | <0 | <0 |
| A-B | <0 | ≥ 0 | ≥ 0 |

# Desbordamiento

- Desbordamiento (Método práctico de detección):El desbordamiento se produce cuando el acarreo generado y el recibido por el último bit son distintos:

| C-2 | | C-2 | |
|---|---|---|---|
| 0110 | (6) | 1001 | (-7) |
| +0100 | (4) | +1000 | (-8) |
| 01010 | (-6) | 10001 | (1) |
| overflow | | overflow | |

# ALU de 32 bits: Desbordamiento

- Desbordamiento (Método práctico de detección):El desbordamiento se produce cuando el acarreo generado y el recibido por el último bit son distintos:

Basta con evaluar $C_{31}$ XOR $C_{32}$

EXTENSIÓN ALU 31.

# ALU de 32 bits: Comparación

- **Slt rd, rs, rt**

rd: 0000 0000 0000 0000 0000 0000 0000 000**r**

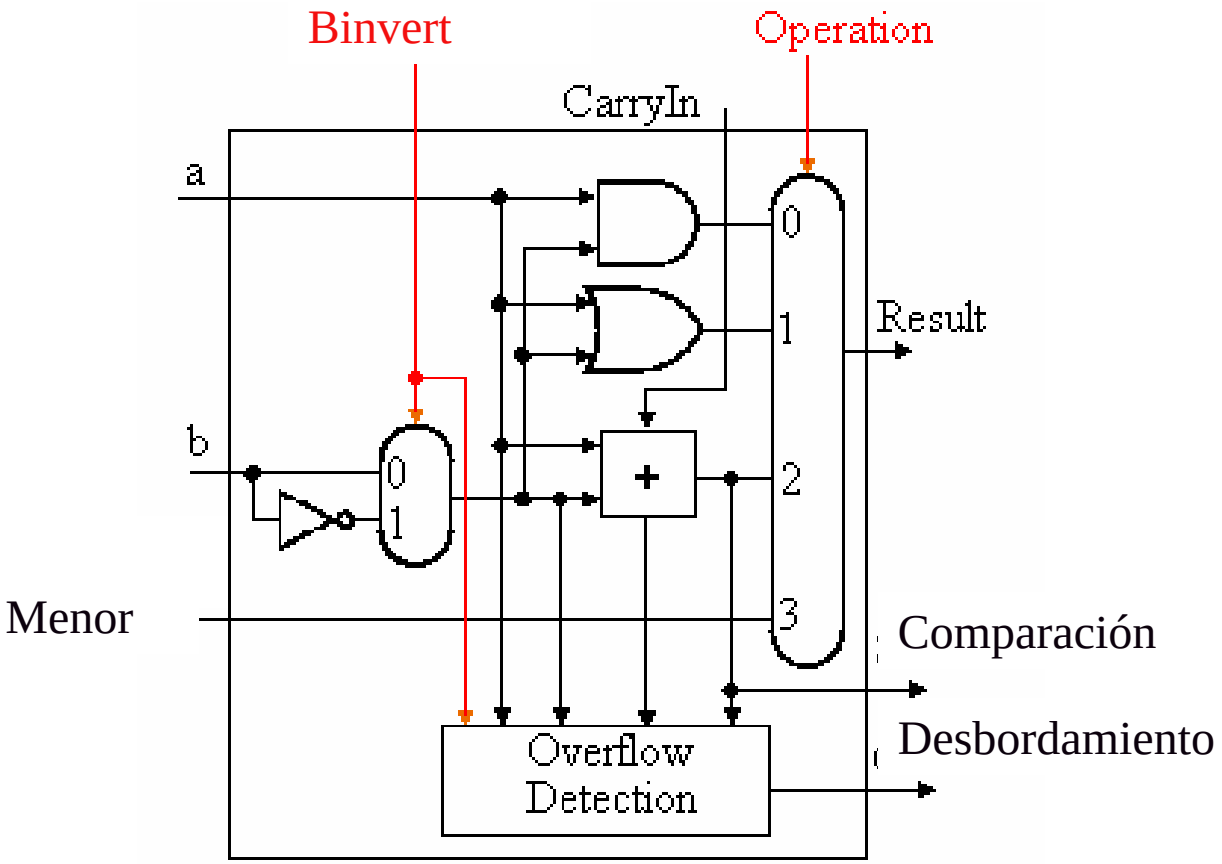
$$r \rightarrow \begin{cases} 1 & \text{if } (rs < rt) \\ 0 & \text{else} \end{cases}$$

- Set on less than:
  - Si a < b : out = 1
  - En otro caso: out = 0
- Implementación: Recoger el signo de a-b→ $C_{32}$

Resultado = [0 0 … 0 $C_{32}$]

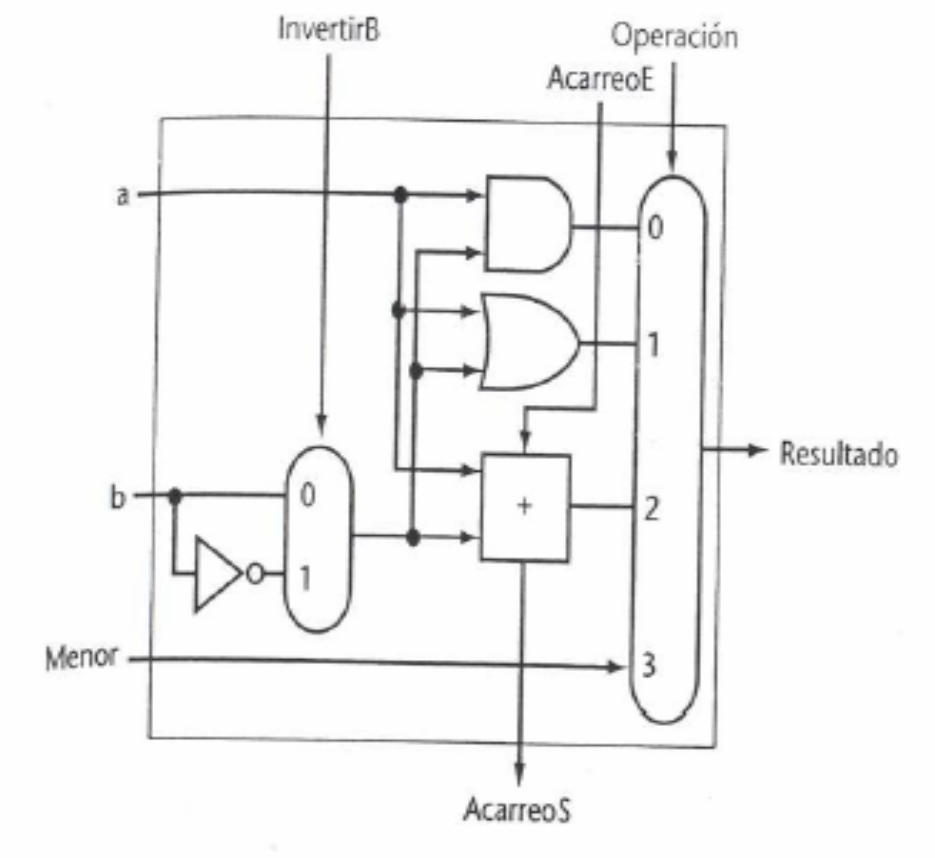
# ALU de 32 bits: AND, OR, suma, resta, desbordamiento, comparación

ALU para el bit más significativo.

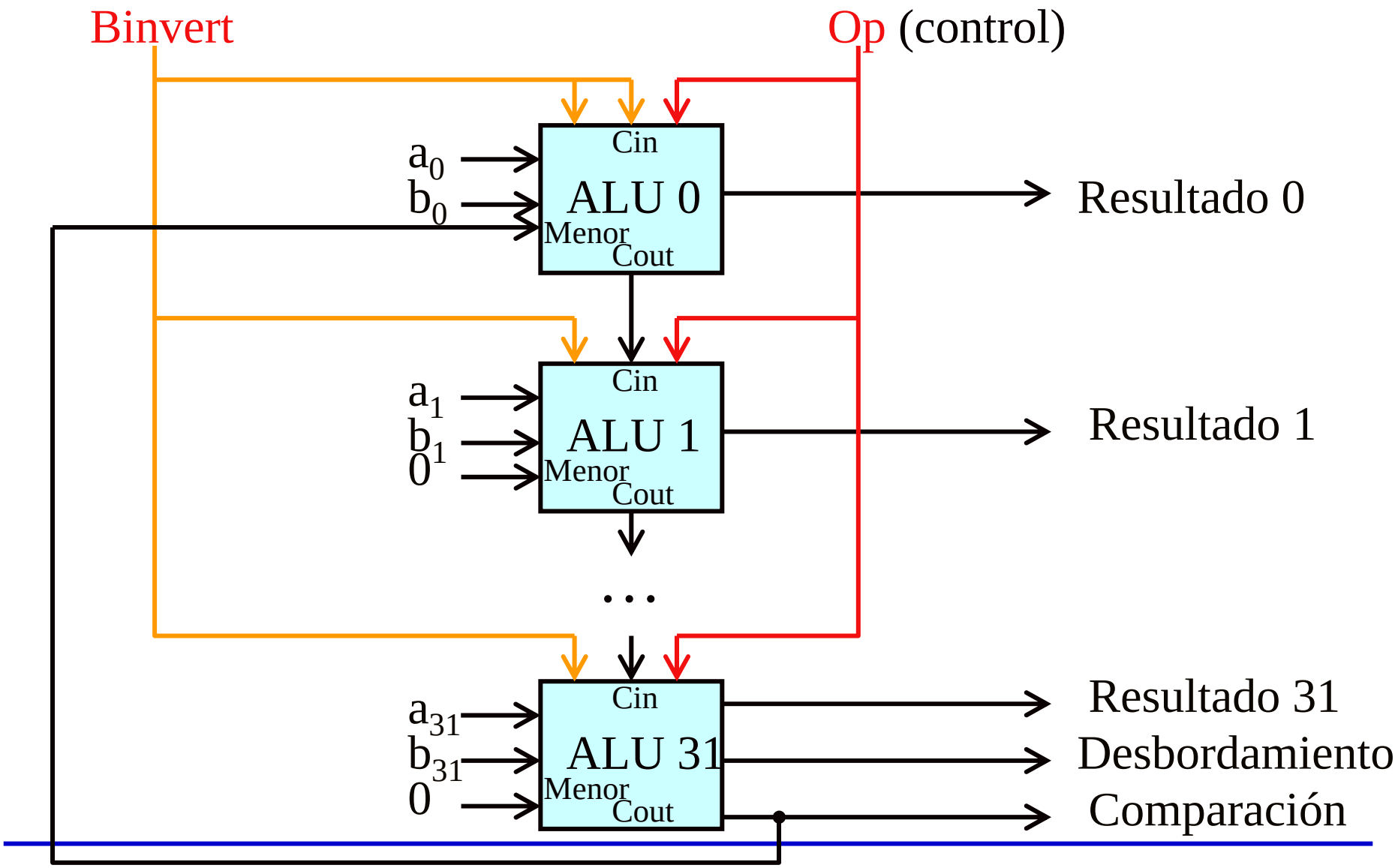


# ALU de 32 bits: AND, OR, suma, resta, desbordamiento, comparación

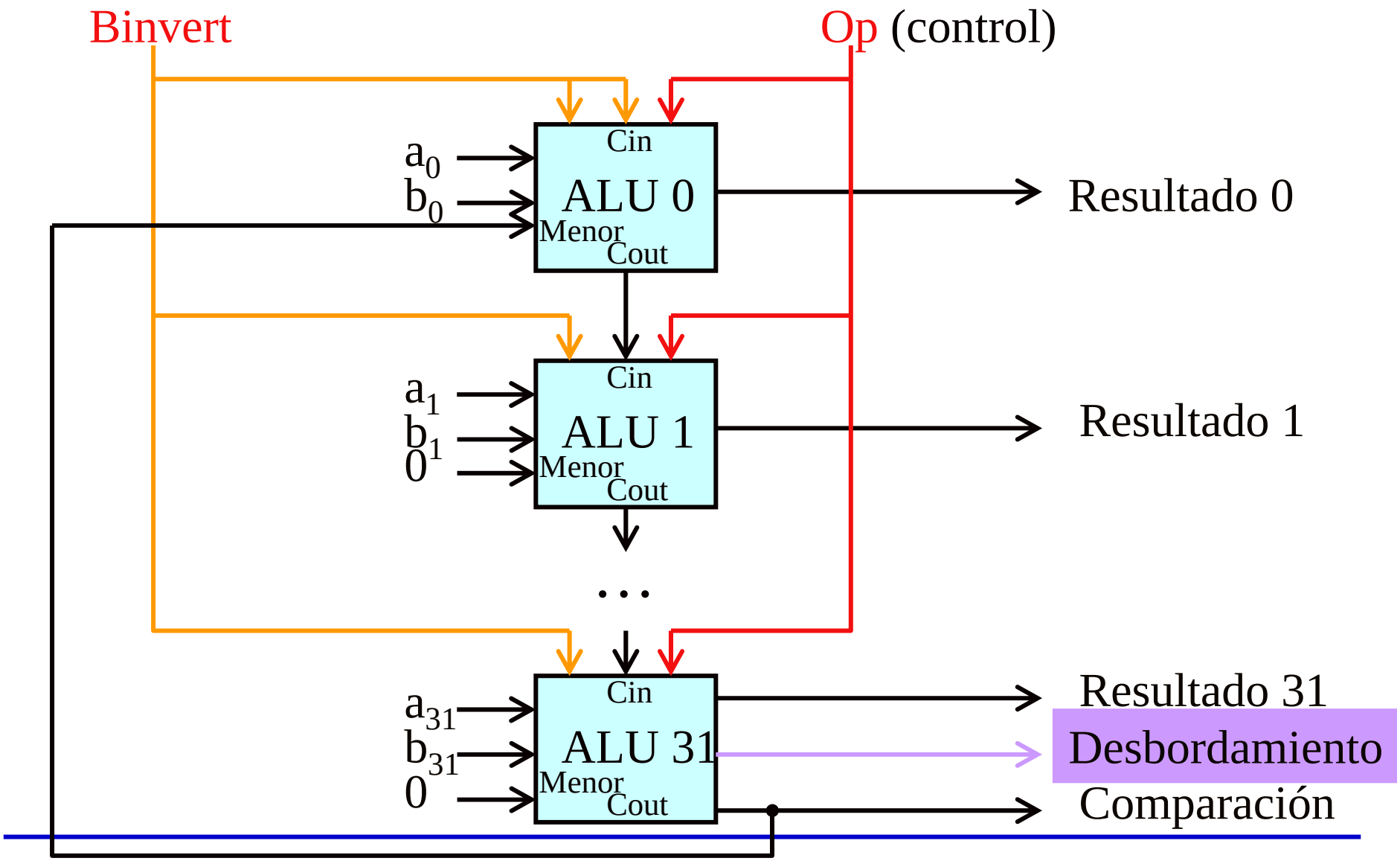
ALU para los otros bits



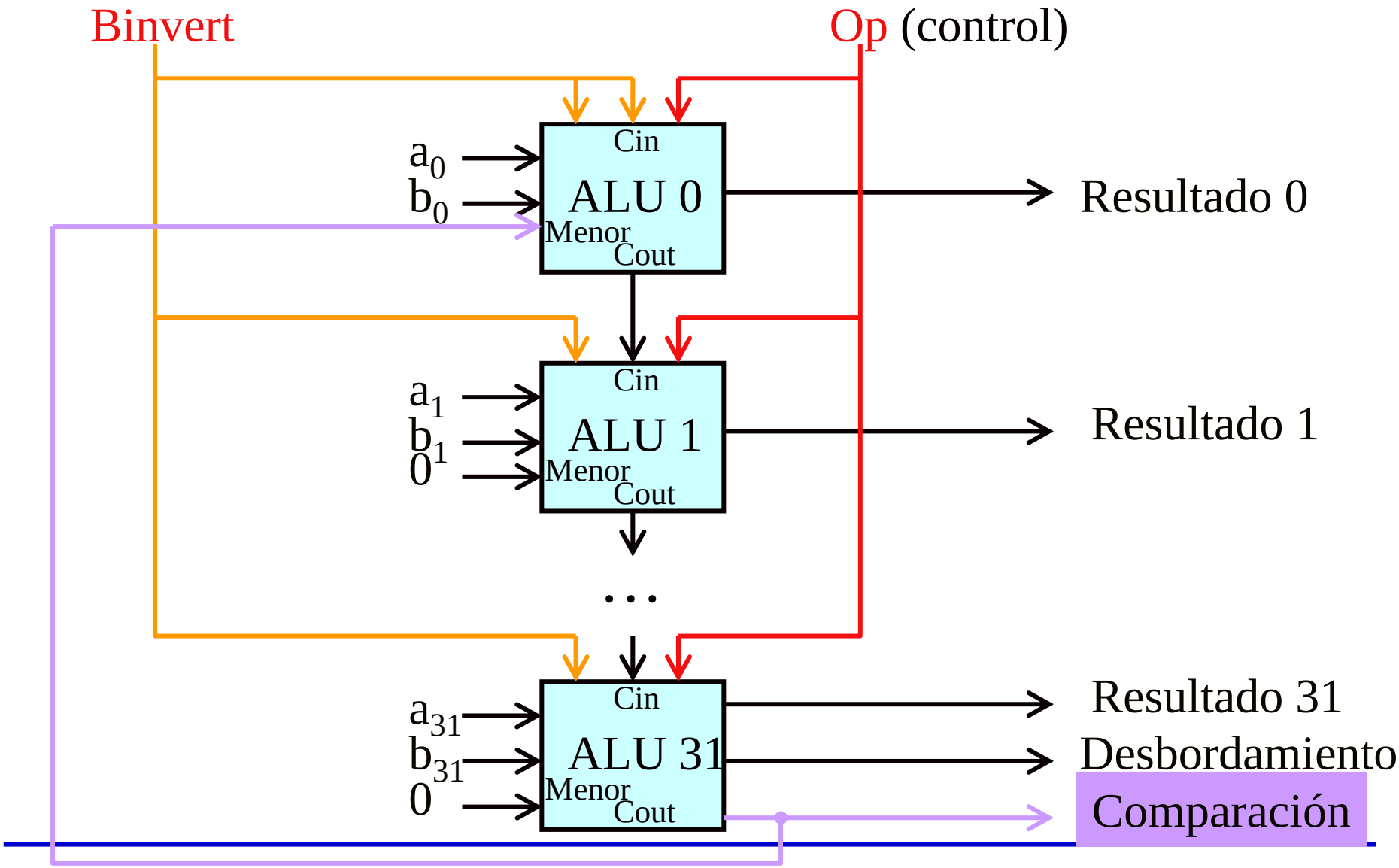
# ALU de 32 bits: AND, OR, suma, resta, desbordamiento, comparación



# ALU de 32 bits: AND, OR, suma, resta, desbordamiento, comparación



# ALU de 32 bits: AND, OR, suma, resta, desbordamiento, comparación

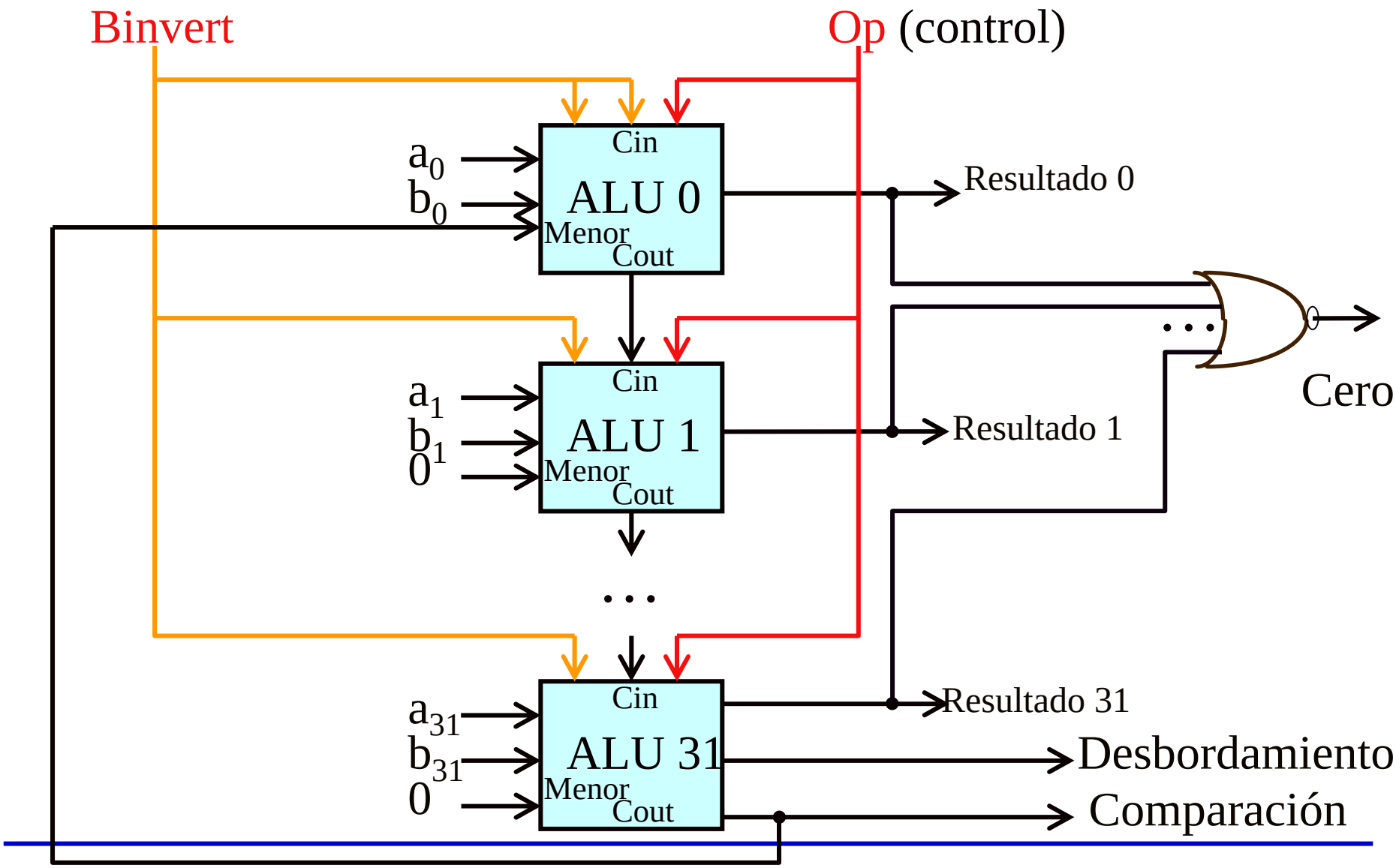


# ALU de 32 bits: Saltos condicionales

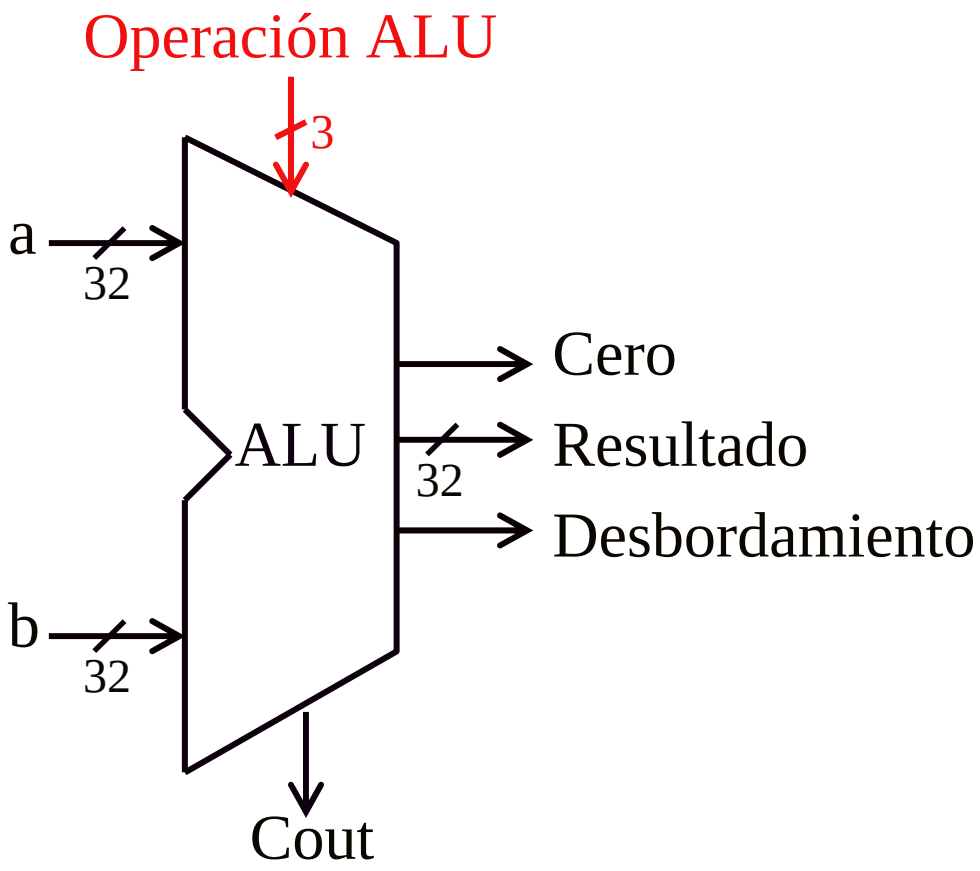
- **beq $t5, $t6, L**
  - Restando: (a-b) = 0 → a = b
  - Añadir hardware para ver si este resultado es 0
  - NOR de los 32 bits de resultado
  - $\overline{\text{Zero} = (\text{Resultado0} + \text{Resultado1} + .. + \text{Resultado31})}$

NOR de los bits de resultado

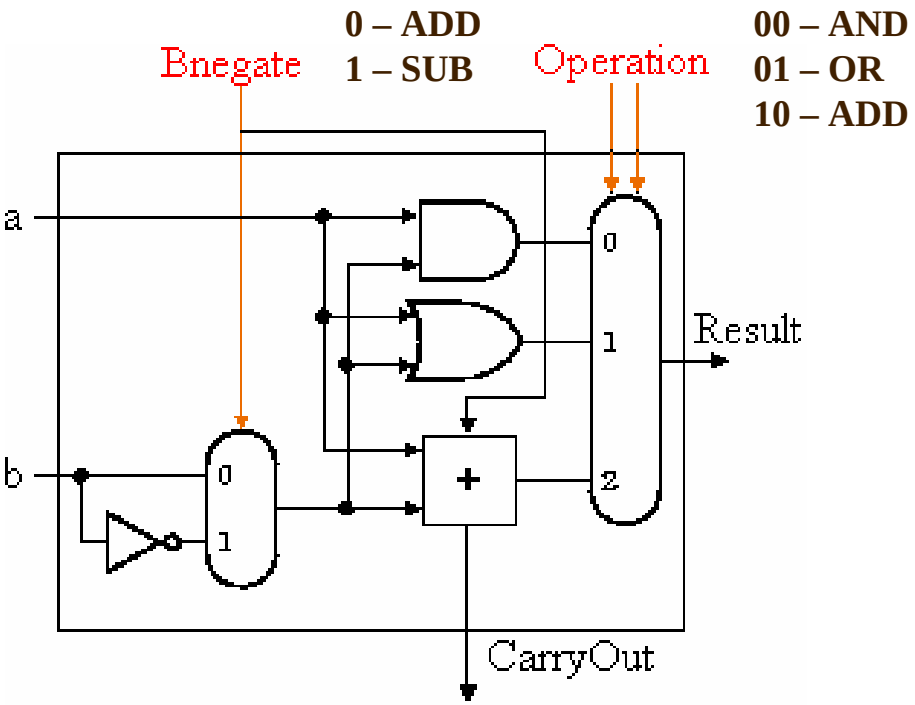
# ALU de 32 bits: AND, OR, suma, resta, desbordamiento, comparación,cero



# ALU de 32 bits: Símbolo universal



# ALU de 32 bits: Señales de control




3 líneas de CONTROL

First bit (LSB)

Operations: AND, OR, ADD, SUB

Control lines: 000 001 010 110

# ALU de 32 bits: Señales de control

| Líneas de control | Función |
|---|---|
| 000 | AND |
| 001 | OR |
| 010 | Suma |
| 110 | Resta |
| 111 | Set on less than |

# Otras operaciones lógicas

Desplazamiento lógico: Desplaza los bits a la izquierda o a la derecha rellenado con cero los bits vacíos

- Implementado fuera de la ALU.

| Operación lógica | Operador en C | Instrucción MIPS |
|---|---|---|
| Despl. Izquierda | << | Sll (Shift Left Logical) |
| Despl. Derecha | >> | Srl (Shift Right Logical) |

Ejemplo: Desplazamiento de 8 posiciones a la izquierda de:

0000 0000 0000 0000 0000 0000 0000 1101

Resultado:

0000 0000 0000 0000 0000 1101 0000 0000

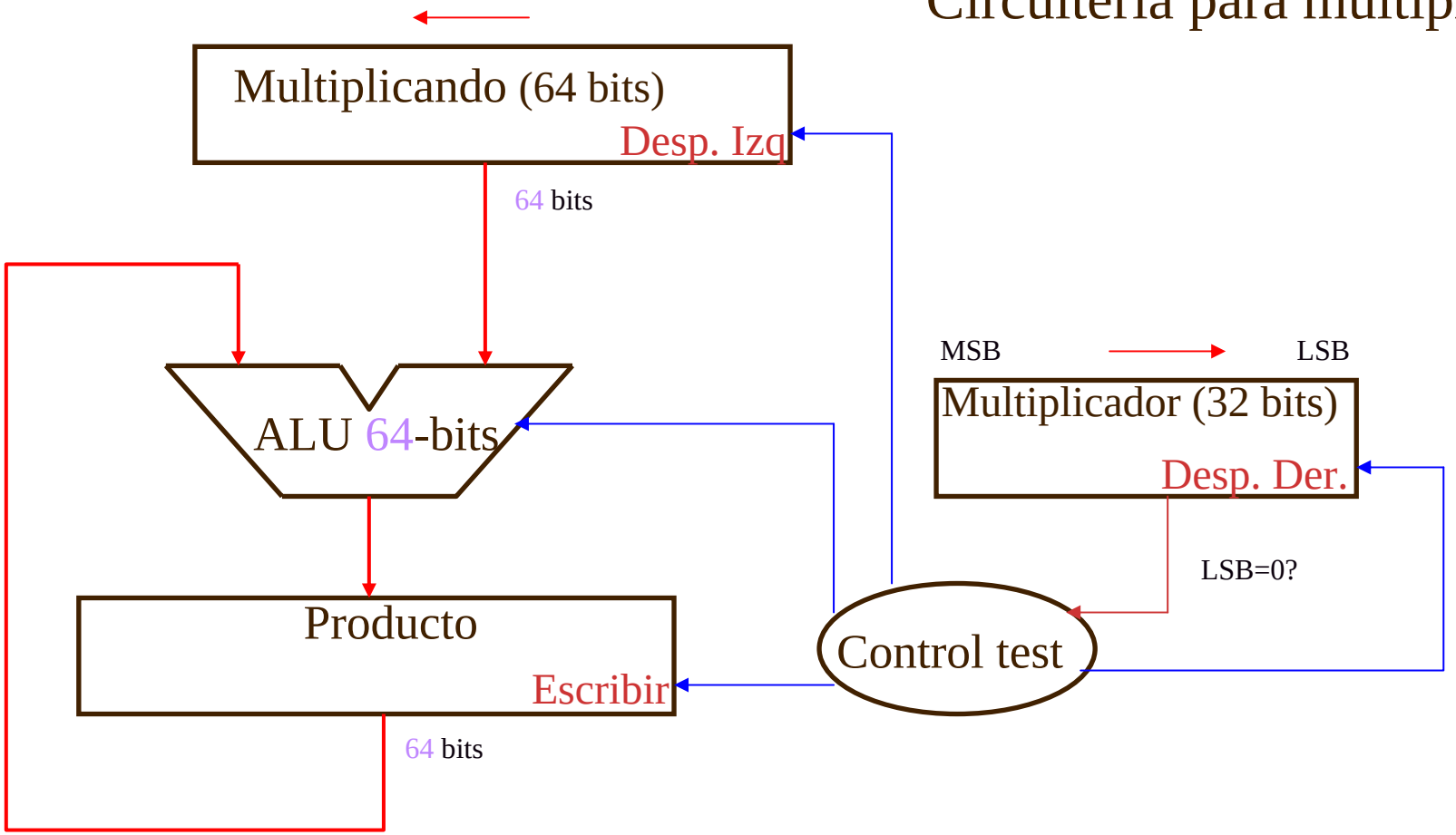
# Multiplicación

- Más compleja que la suma
  - Implementada con: Sumas y desplazamientos.
  - Coste área y tiempo
- 3 versiones del algoritmo lápiz y papel:

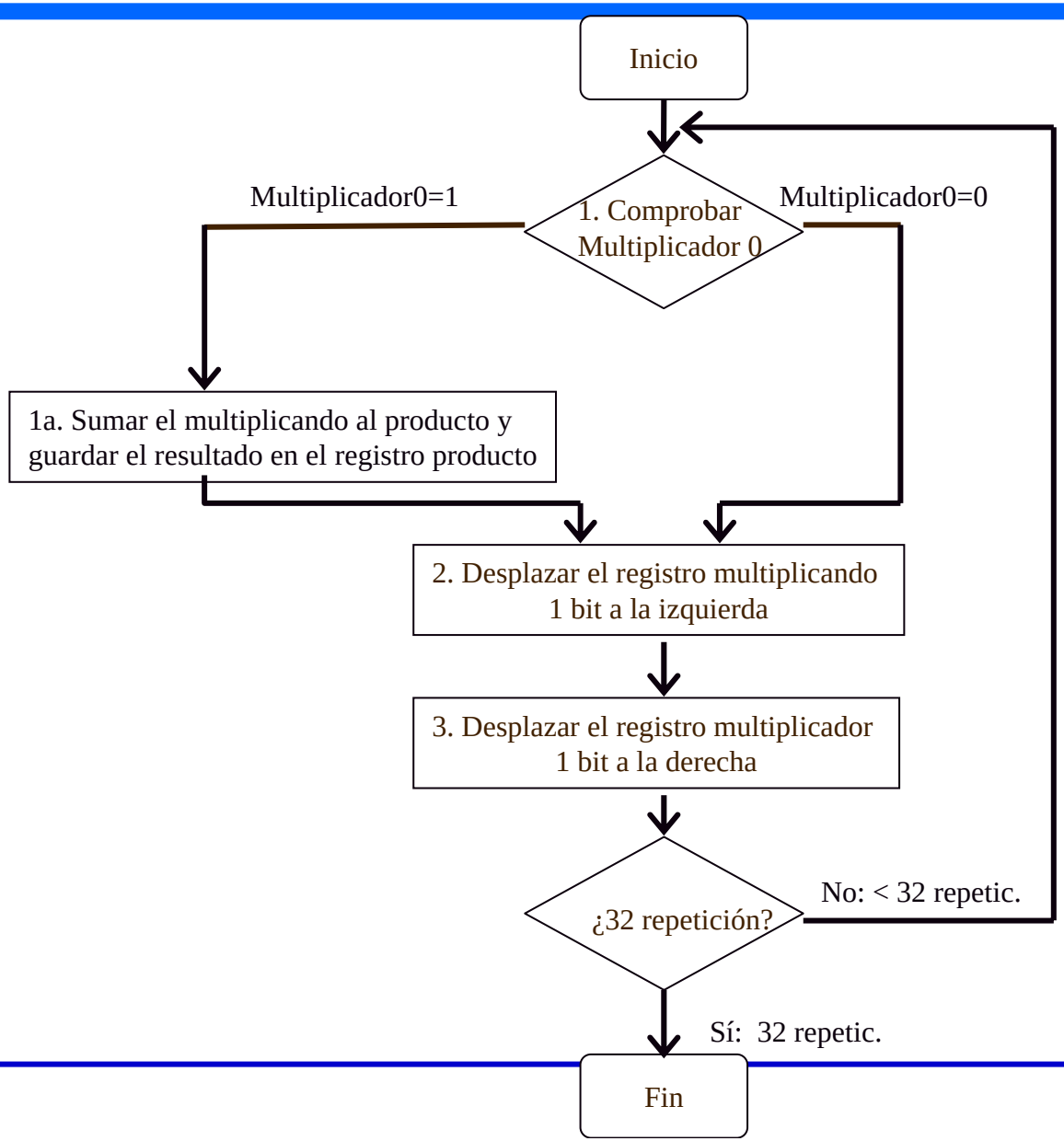
```
    0010      (multiplicando)
  x 1011      (multiplicador)
    0010      1 -> copia & shift
   0010       1 -> copia & shift
  0000        0 -> shift
 0010         1 -> copia & shift
00010110      Suma Productos parciales
```

# Multiplicación: Primera versión

Circuitería para multiplicar



# Multiplicación: Primera versión

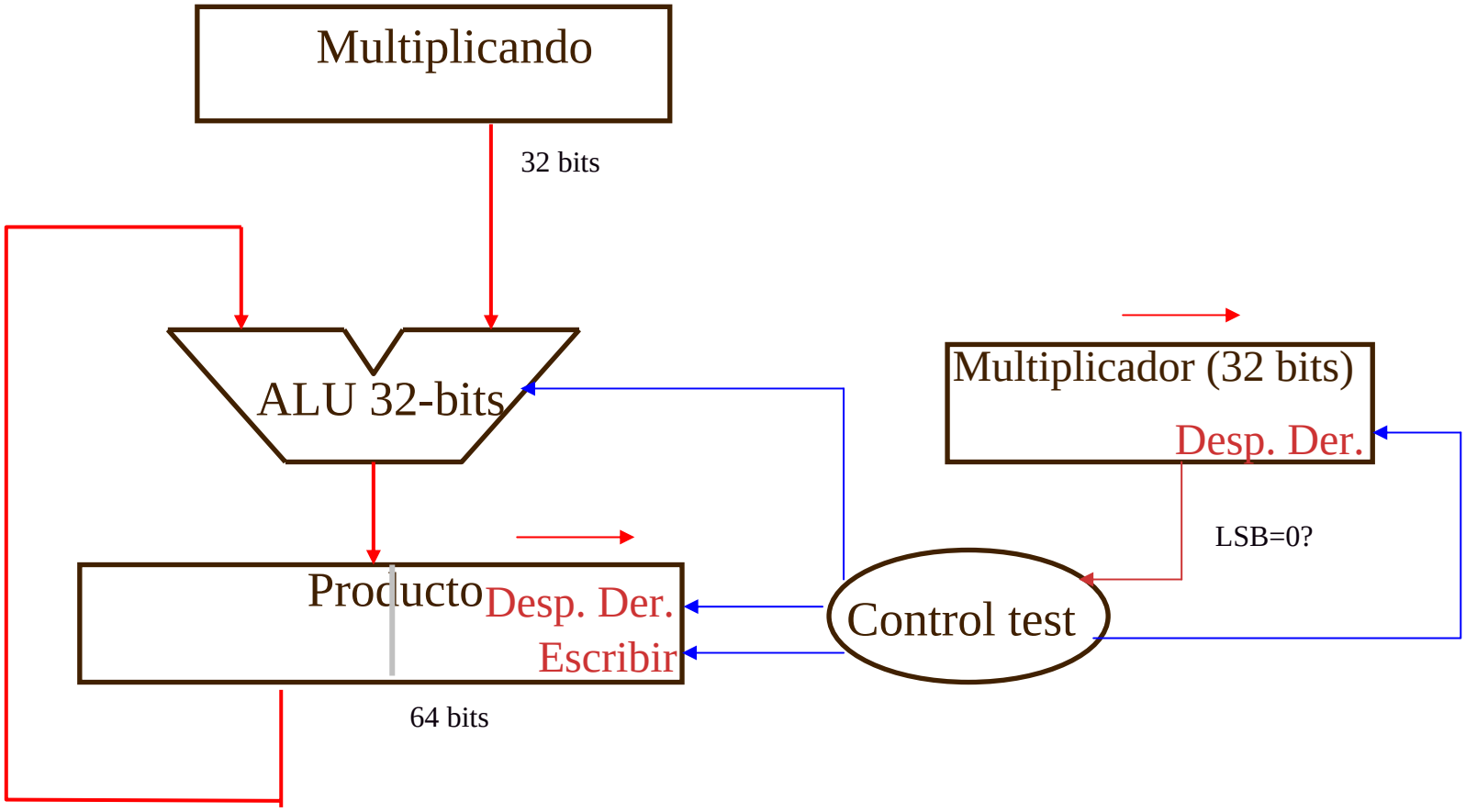


# Multiplicación: Primera versión

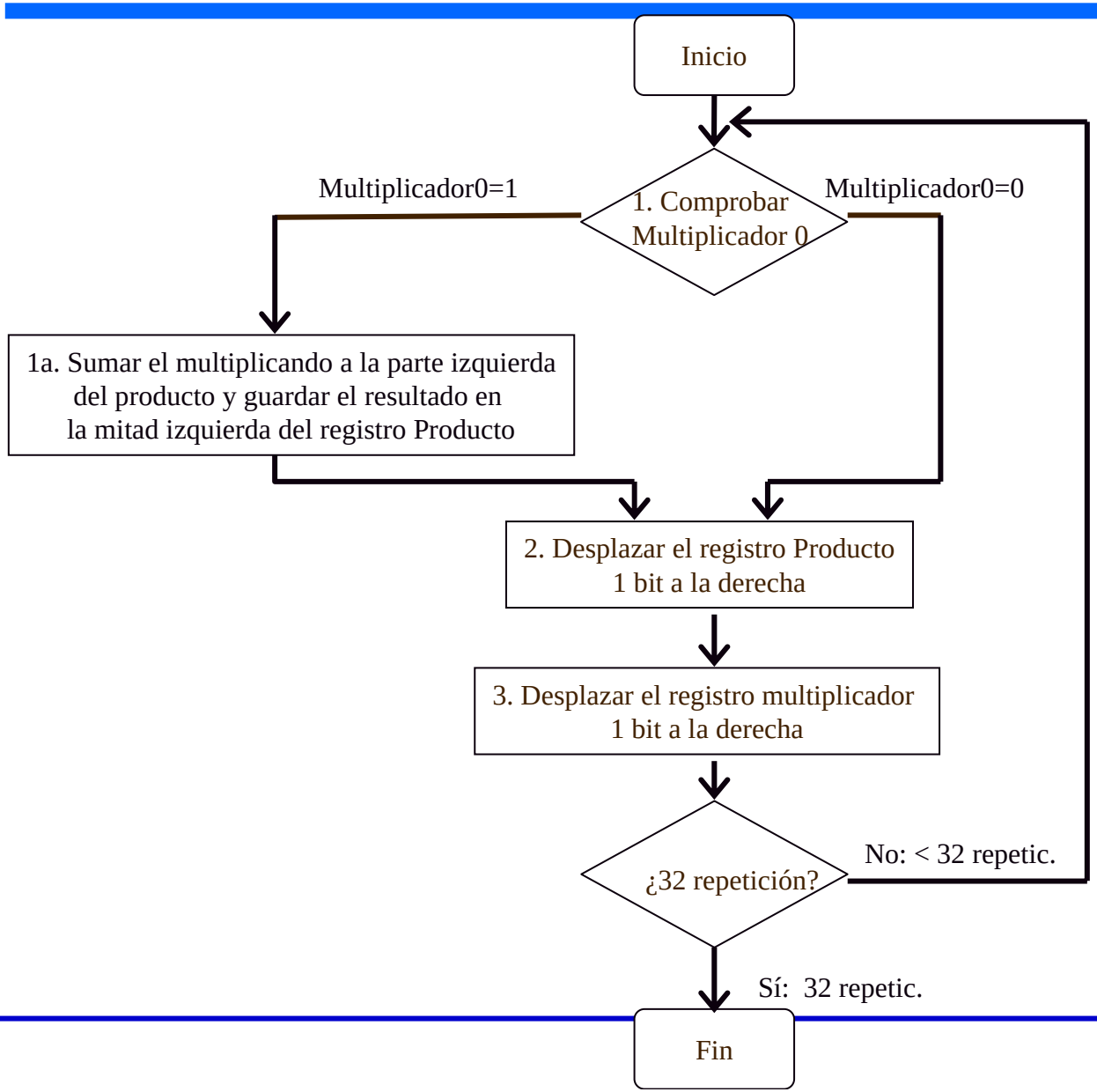
| Iteración | Paso | Multiplicador | Multiplicando | Producto |
|---|---|---|---|---|
| 0 | Valores iniciales | 0011 | 0000 0010 | 0000 0000 |
| 1 | 1a: 1→ Prod=Prod+Mcando | 0011 | 0000 0010 | 0000 0010 |
| | 2: Desplazar el Mcando a la izquierda | 0011 | 0000 0100 | 0000 0010 |
| | 3: Desplazar el Mcador a la derecha | 0001 | 0000 0100 | 0000 0010 |
| 2 | 1a: 1→ Prod=Prod+Mcando | 0001 | 0000 0100 | 0000 0110 |
| | 2: Desplazar el Mcando a la izquierda | 0001 | 0000 1000 | 0000 0110 |
| | 3: Desplazar el Mcador a la derecha | 0000 | 0000 1000 | 0000 0110 |
| 3 | 1a: 0→ Ninguna operación | 0000 | 0000 1000 | 0000 0110 |
| | 2: Desplazar el Mcando a la izquierda | 0000 | 0001 0000 | 0000 0110 |
| | 3: Desplazar el Mcador a la derecha | 0000 | 0001 0000 | 0000 0110 |
| 4 | 1a: 0→ Ninguna operación | 0000 | 0001 0000 | 0000 0110 |
| | 2: Desplazar el Mcando a la izquierda | 0000 | 0010 0000 | 0000 0110 |
| | 3: Desplazar el Mcador a la derecha | 0000 | 0010 0000 | 0000 0110 |

# Multiplicación: Segunda versión

Circuitería para multiplicar



# Multiplicación: Segunda versión

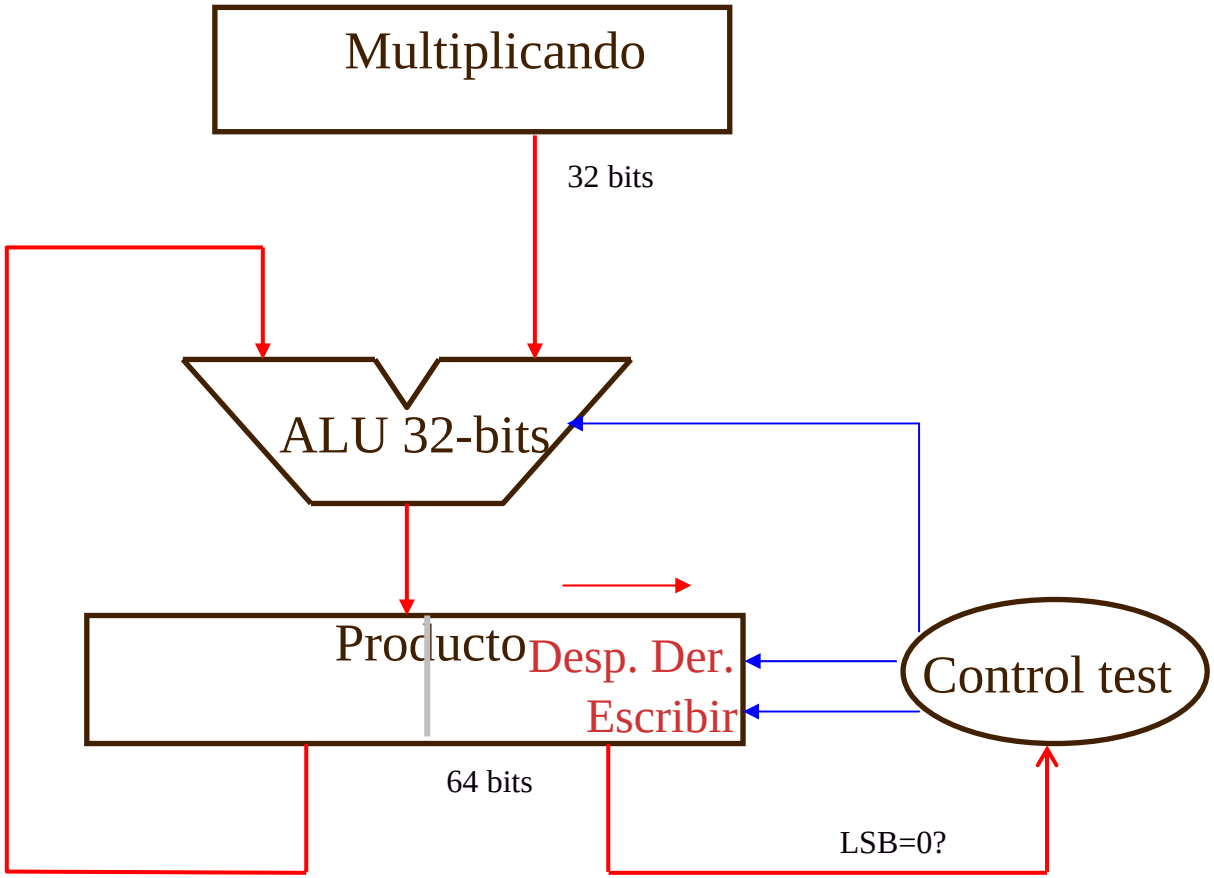


# Multiplicación: Segunda versión

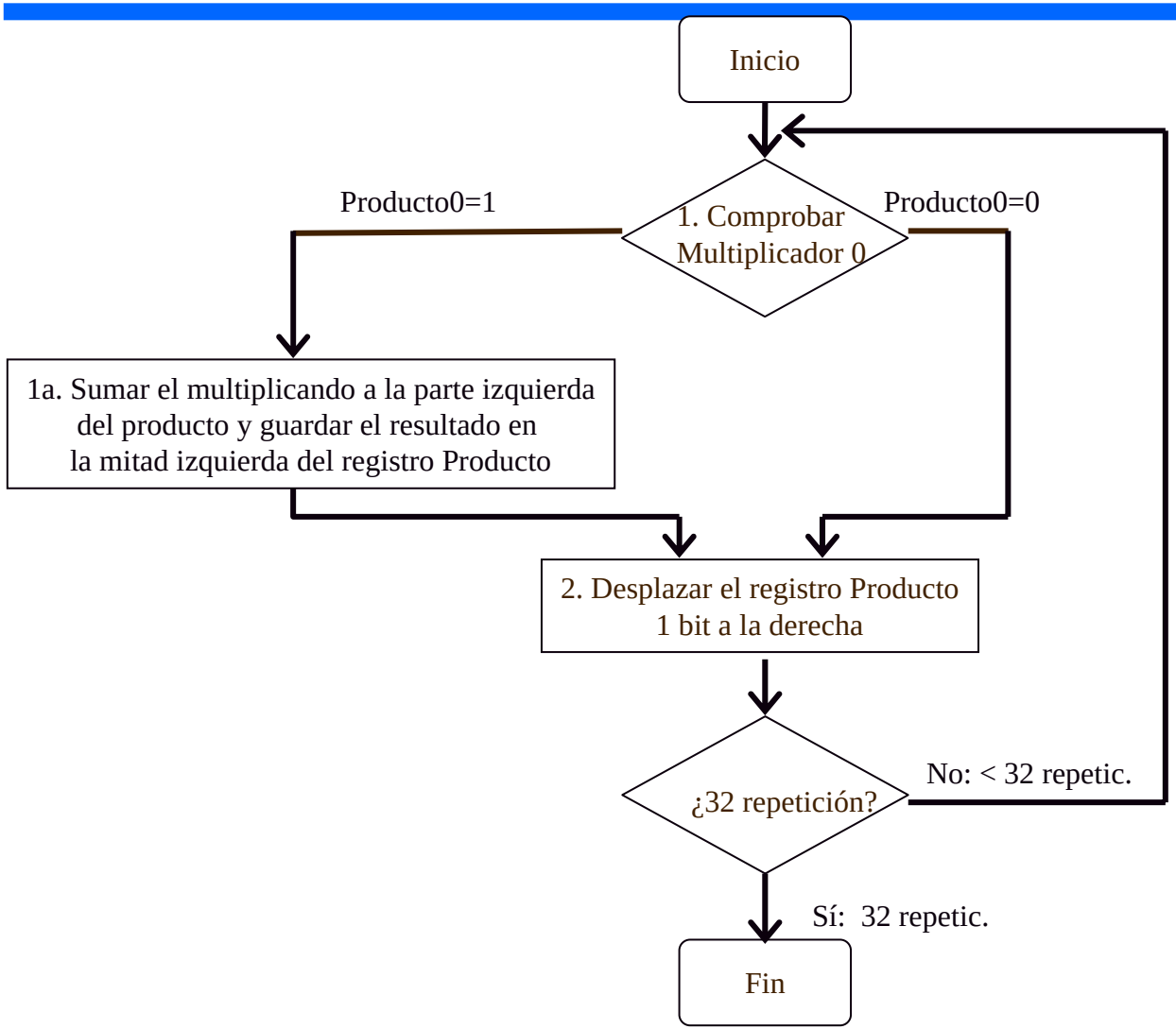
| Iteración | Paso | Multiplicador | Multiplicando | Producto |
|---|---|---|---|---|
| 0 | Valores iniciales | 0011 | 0010 | 0000 0000 |
| 1 | 1a: 1→ Prod=Prod+Mcando | 0011 | 0010 | 0010 0000 |
| | 2: Desplazar el Producto a la derecha | 0011 | 0010 | 0001 0000 |
| | 3: Desplazar el Mcador a la derecha | 0001 | 0010 | 0001 0000 |
| 2 | 1a: 1→ Prod=Prod+Mcando | 0001 | 0010 | 0011 0000 |
| | 2: Desplazar el Producto a la derecha | 0001 | 0010 | 0001 1000 |
| | 3: Desplazar el Mcador a la derecha | 0000 | 0010 | 0001 1000 |
| 3 | 1a: 0→ Ninguna operación | 0000 | 0010 | 0001 1000 |
| | 2: Desplazar el Producto a la derecha | 0000 | 0010 | 0000 1100 |
| | 3: Desplazar el Mcador a la derecha | 0000 | 0010 | 0000 1100 |
| 4 | 1a: 0→ Ninguna operación | 0000 | 0010 | 0000 1100 |
| | 2: Desplazar el Producto a la derecha | 0000 | 0010 | 0000 0110 |
| | 3: Desplazar el Mcador a la derecha | 0000 | 0010 | 0000 0110 |

# Multiplicación: Tercera versión

Circuitería para multiplicar



# Multiplicación: Versión Final



# Multiplicación: Versión Final

| Iteración | Paso | Multiplicando | Producto |
|---|---|---|---|
| 0 | Valores iniciales | 0010 | 0000 0011 |
| 1 | 1a: 1→ Prod=Prod+Mcando | 0010 | 0010 0011 |
| | 2: Desplazar el Producto a la derecha | 0010 | 0001 0001 |
| 2 | 1a: 1→ Prod=Prod+Mcando | 0010 | 0011 0001 |
| | 2: Desplazar el Producto a la derecha | 0010 | 0001 1000 |
| 3 | 1a: 0→ Ninguna operación | 0010 | 0001 1100 |
| | 2: Desplazar el Producto a la derecha | 0010 | 0000 1100 |
| 4 | 1a: 0→ Ninguna operación | 0010 | 0000 1100 |
| | 2: Desplazar el Producto a la derecha | 0010 | 0000 0110 |

# Multiplicación con signo

- Multiplicadores vistos hasta ahora: números sin signo.
- Para números con signo: no funcionan
- Posibilidad:
  - Transformar los operandos a positivos.
  - Multiplicar los valores positivos.
  - Negar el resultado (si es necesario).

$$|c| = |a| \cdot |b|$$
$$\text{signo}(c) = \text{signo}(a) \text{ XOR } \text{signo}(b)$$

Optimizable→ Algoritmo de Booth

# Multiplicación con signo: Algoritmo de Booth

- CLAVE: RECODIFICACION DEL MULTIPLICADOR (factorización inteligente de la representación en complemento a dos del multiplicador)
- Observación Booth (para números positivos):

  0 0 0 1 1 1 1 0 (un bloque de 1s rodeado de 0s)

  M x (0 0 0 1 1 1 1 0) = M x ($2^4 + 2^3 + 2^2 + 2^1$)=

  = M x 30
- Nótese: $\mathbf{2^n + 2^{n-1} + \ldots + 2^{n-k} = 2^{n+1} - 2^{n-k}}$

  => M x (0 0 0 1 1 1 1 0) = M x ($2^5 - 2^1$)
- Algoritmo de Booth:
  - Restar cuando comienza una cadena de unos (1-0)
  - Sumar cuando finaliza una cadena de unos (0-1)

# Multiplicación con signo: Algoritmo de Booth

- Recodificación del multiplicador:
  1. Procesar desde el LSB al MSB. Reemplazar cada 0 con un 0.
  2. Cuando se encuentra el primer 1, insertar un -1 en la posición del número recodificado. Saltar todos los 1s del bloque hasta localizar el siguiente
  3. Reemplazar el primer 0 con un 1. Si se llega al MSB sin encontrar ningún 0, no hacer nada.

| Bit en curso | Bit de la derecha | Codificación | Explicación |
|---|---|---|---|
| 0 | 0 | 0 | Nada |
| 0 | 1 | 1 | Sumar Multiplicando |
| 1 | 0 | -1 | Restar Multiplicando |
| 1 | 1 | 0 | Nada |

# Multiplicación con signo: Algoritmo de Booth

- Recodificación del multiplicador. Ejemplo:

$$0011111011001 = 512 + 256 + 128 + 64 + 16 + 8 + 1 = 985$$

$$0100\bar{1}10\bar{1}01\bar{1} = 1024 - 64 + 32 - 8 + 2 - 1 = 985$$

Fácil adaptación de la circuitería de multiplicación para incluir el algoritmo de Booth: Análisis de dos bits

# Multiplicación con signo: Algoritmo de Booth

| Valores iniciales | Paso | Multiplicando | Producto |
|---|---|---|---|
| 0 | Valores iniciales | 0010 | 0000 1101 0 |
| 1 | 10→ Prod=Prod-Mcando | 0010 | 1110 1101 0 |
| | Desplazar el Producto a la derecha | 0010 | 1111 0110 1 |
| 2 | 01→ Prod=Prod+Mcando | 0010 | 0001 0110 1 |
| | Desplazar el Producto a la derecha | 0010 | 0000 1011 0 |
| 3 | 10→ Prod=Prod-Mcando | 0010 | 1110 1011 0 |
| | Desplazar el Producto a la derecha | 0010 | 1111 0101 1 |
| 4 | 11→ Ninguna operación | 0010 | 1111 0101 1 |
| | Desplazar el Producto a la derecha | 0010 | 1111 1010 1 |

# Multiplicación en el MIPS

- Utilización del hardware existente: ALU y desplazadores
- Registros de propósito general para almacenar el producto de 64 bits (Hi, Lo)
- Dos instrucciones de multiplicación:
  - Mult: con signo (→ Booth)
  - Multu: sin signo
- Dos instrucciones para mover los contenidos de Hi y Lo a registros de propósito general: mflo, mfhi
- *No hay hardware para detección de overflow*

=> Detección con software

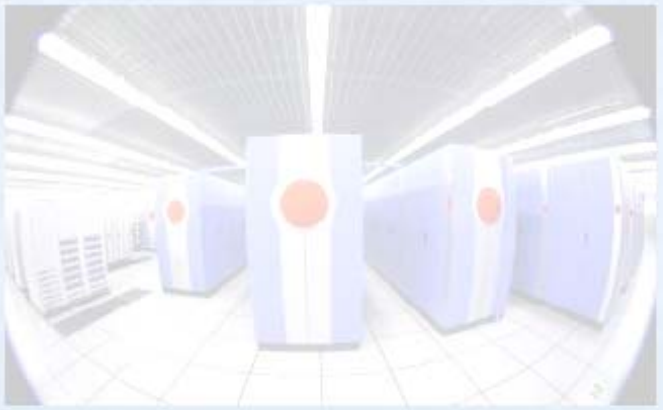
# División en el MIPS

- Hardware adicional: registros de 64-bits capaces de SLL/SRA
  - Hi contiene el resto (mfhi)
  - Lo contiene el cociente (mflo)
- Instructiones
  - Div: división con signo
  - Divu: división sin signo
- *No hay hardware para detección de overflow*

  ⇒ Detección con software
- División por cero: chequeada con software

# Punto flotante (PF)



# Operaciones en punto flotante

1. Suma
2. Multiplicación
3. División

Las estudiaremos en este curso

# Aritmética en punto flotante

- Las operaciones aritméticas en punto flotante tienen una implementación compleja por dos motivos principalmente:
  - Alineamiento de la mantisa
  - Redondeo
- Paradójicamente la suma presenta más problemas que la multiplicación
- Los sistemas más sencillos no soportan aritmética en punto flotante
  - Por su elevado coste y consumo de potencia
  - Las operaciones en punto flotante se realizan en punto fijo
- Los microprocesadores que habitualmente utilizamos soportan todo tipo de operaciones en punto flotante
    - Simple y doble precisión
    - Exponenciación, logaritmos, trigonométricas,…

# SUMA: Suma y resta PF no son asociativas

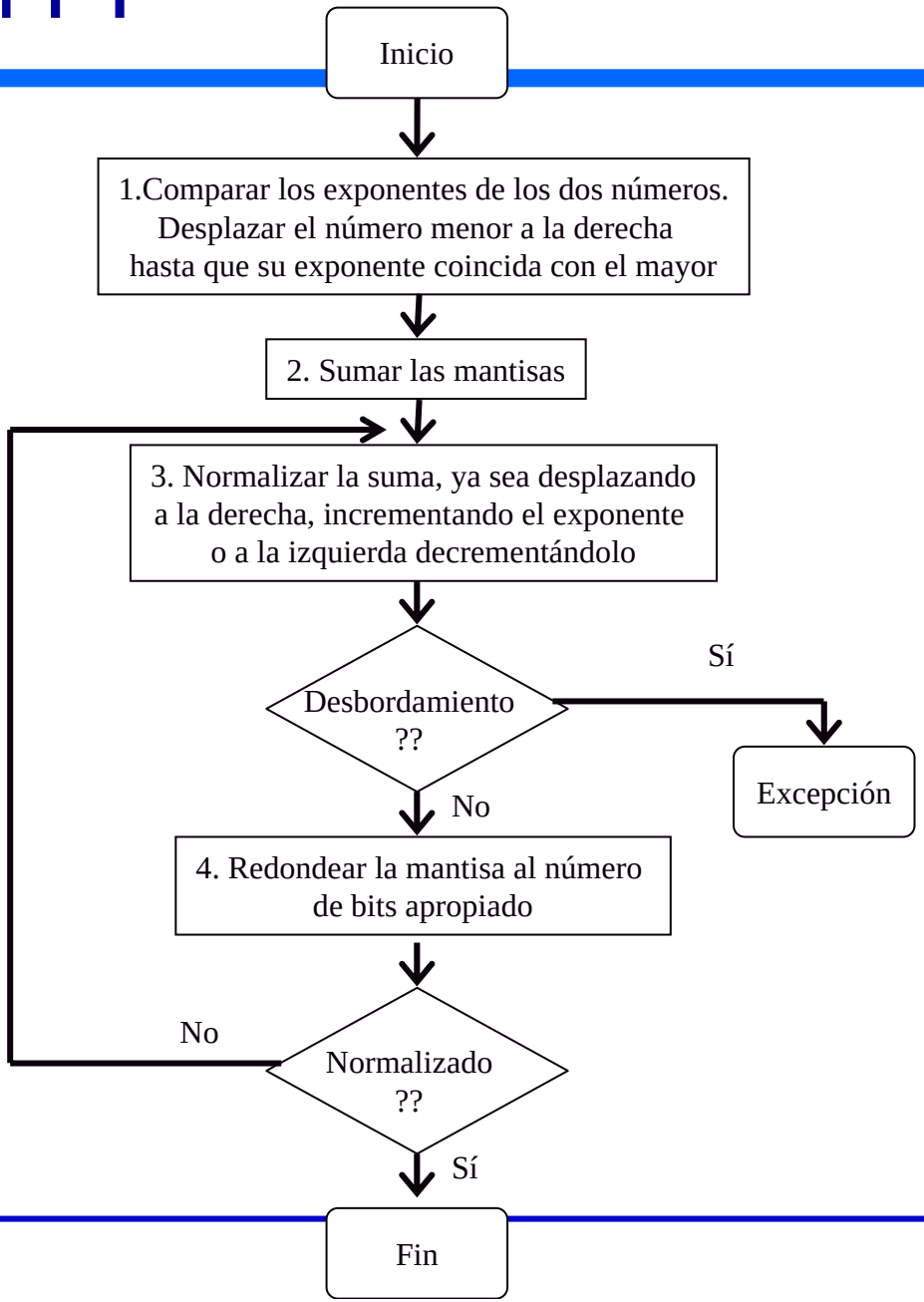
## Suma y resta en PF no son asociativas:

- $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
- $x + (y + z) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0)$
  $= -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) = \underline{0.0}$
- $(x + y) + z = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0$
  $= (0.0) + 1.0 = \underline{1.0}$

## Motivo:

- El resultado PF sólo aproxima el resultado real
- $1.5 \times 10^{38}$ es mucho más grande que 1.0 → $1.5 \times 10^{38} + 1.0 \sim 1.5 \times 10^{38}$

# Suma en PF


Inicio
1.Comparar los exponentes de los dos números. Desplazar el número menor a la derecha hasta que su exponente coincida con el mayor
2. Sumar las mantisas
3. Normalizar la suma, ya sea desplazando a la derecha, incrementando el exponente o a la izquierda decrementándolo
Desbordamiento ??
Sí
Excepción
No
4. Redondear la mantisa al número de bits apropiado
Normalizado ??
No
Sí
Fin

# Suma en PF

**1.5e2 + 1.3e4 = 0.015e4 + 1.3e4 = 1.315e4**

La alineación es el elemento clave

- Es necesario alinear los números para que tengan el mismo exponente

**Ejemplo**

**A: 0000010110001010001001010100011 0**

**B: 0000001111001000011100000011110 1**

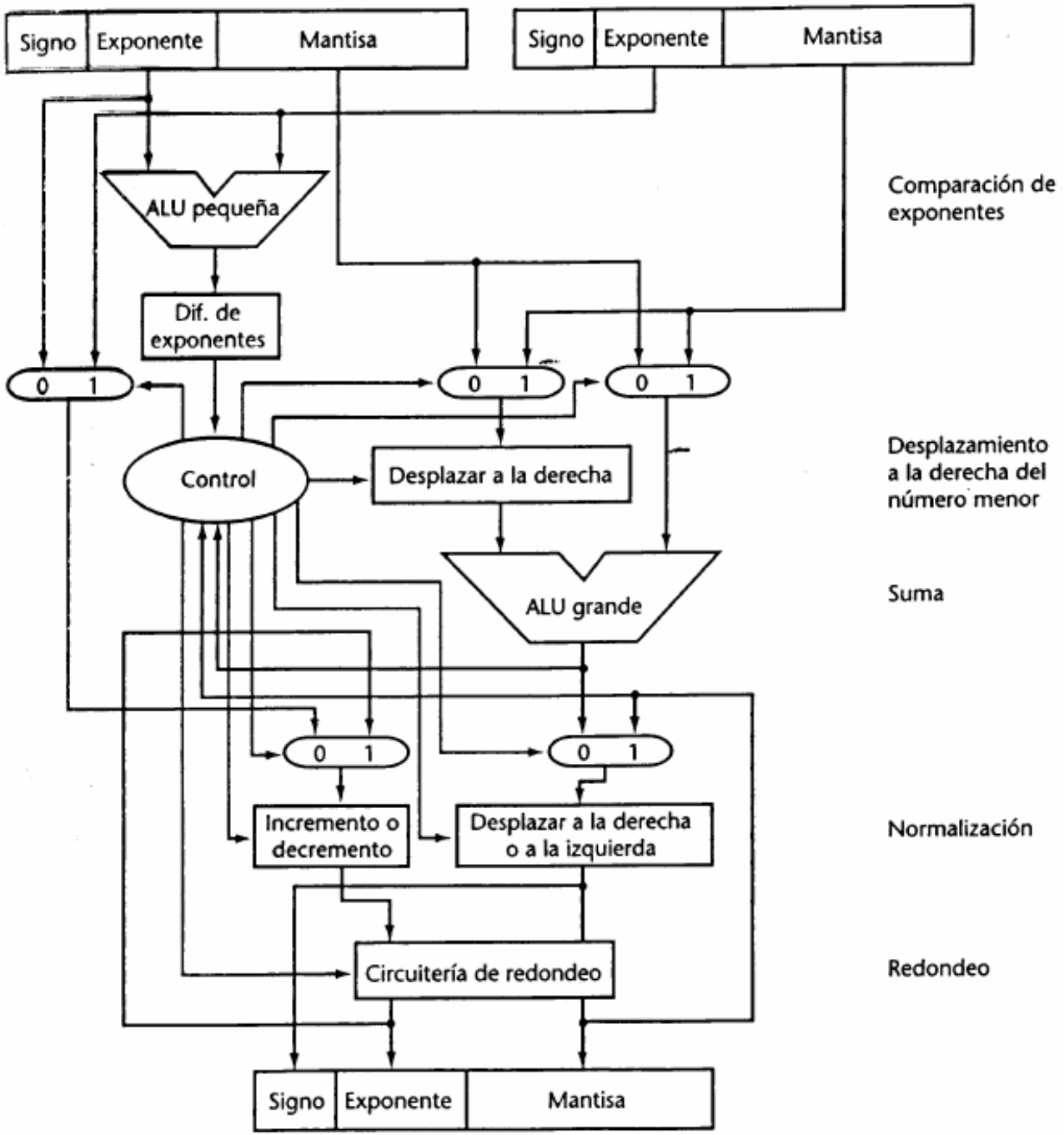
**Exponente A = 11 (=-116)**

**Exponente B = 7 (=-120)**

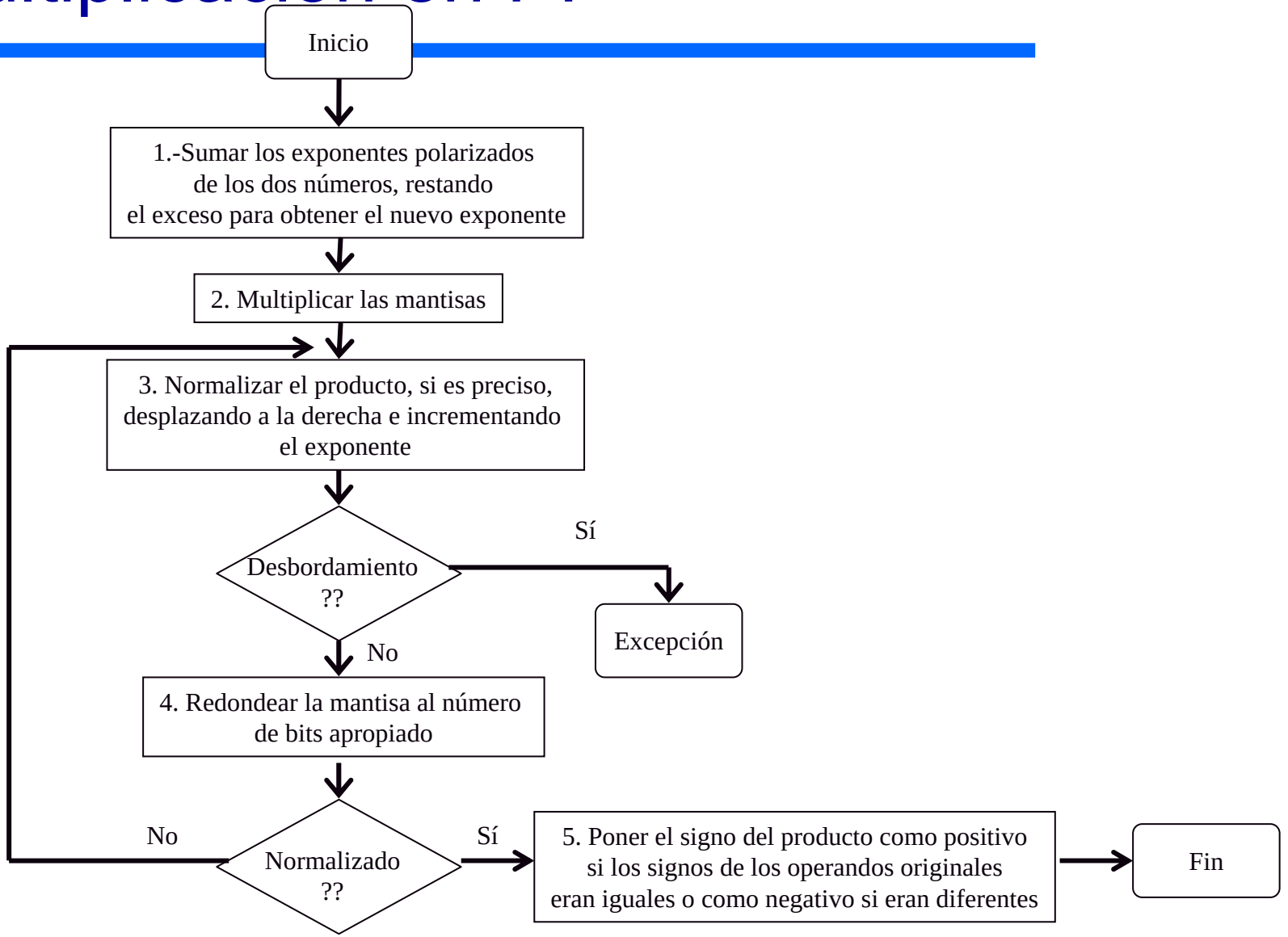
```
      1.00010100010010101000110
+     0.00011001000011100000011101
-------------------------------------
      1.00101101010110001001011101
```

**R: 0000010110010110101011000100100 1**

# Suma PF


Signo
Exponente
Mantisa
Signo
Exponente
Mantisa
ALU pequeña
Comparación de exponentes
Dif. de exponentes
0 1
0 1
0 1
Control
Desplazar a la derecha
Desplazamiento a la derecha del número menor
ALU grande
Suma
0 1
0 1
Incremento o decremento
Desplazar a la derecha o a la izquierda
Normalización
Circuitería de redondeo
Redondeo
Signo
Exponente
Mantisa

# Multiplicación en PF



# Multiplicación en PF

$A = m_A \, 2^{eA}$

$B = m_B \, 2^{eB}$

$A \times B = m_A \times m_B \; 2^{(eA+eB)}$

**A: 0100010110001010001001010101000110**

**B: 0100011111001000011110000000111101**

**Exponentes:**

**Exponente A = 139 (=12)**

**Exponente B = 71 (=-56) => Exponente = 83 (=-44)**

# Multiplicación en punto flotante

**A: 01000101100010100010010101000110**
**B: 00100011110010000111000000111101**

**Producto de mantisas:**

**1.00010100010010101000110**
**1.10010000111000000111101 x**

---

**1.10110000101001101011111*1010011000000110101110***

**Resultado = 00101001110110000101001101011111**

# Punto flotante en el MIPS

- **Elementos:**

| Nombre | Ejemplo |
|---|---|
| 32 registros de coma flotante | $f0, $f1, …$f31 |
| 230 palabras de memoria | Memoria[0]<br>Memoria[4]<br>…<br>Memoria[4294967292] |

# Punto flotante en el MIPS

| Categoría | Instrucción | Ejemplo | Significado |
|---|---|---|---|
| Aritmética | FP add single | add.s $f2, $f4, $f6 | $f2 = $f4 + $f6 |
| | FP substract single | sub.s $f2, $f4, $f6 | $f2 = $f4 - $f6 |
| | FP multiply single | mul.s $f2, $f4, $f6 | $f2 = $f4 x $f6 |
| | FP divide single | div.s $f2, $f4, $f6 | $f2 = $f4 / $f6 |
| | FP add double | add.d $f2, $f4, $f6 | $f2 = $f4 + $f6 |
| | FP substract double | sub.d $f2, $f4, $f6 | $f2 = $f4 - $f6 |
| | FP multiply double | mul.d $f2, $f4, $f6 | $f2 = $f4 x $f6 |
| | FP divide double | div.d $f2, $f4, $f6 | $f2 = $f4 / $f6 |
| Transferencia de datos | load word copr. 1 | lwc1 $f1, 100($s2) | $f1 = Mem[$s2+100] |
| | store word copr. 1 | swc1 $f1,100($s2) | Mem[$s2+100]= $f1 |
| Salto condicional | branch on FP true | bclt 25 | si (cond==1) ir a PC + 4 +100 |
| | branch on FP false | bclf 25 | si (cond == 0) ir a PC + 4 + 100 |
| | FP compare single (eq,ne,lt,le,gt,ge) | c.lt.s $f2, $f4 | si ($f2 < $f4) cond =1; si no cond = 0; |
| | FP compare double (eq,ne,lt,le,gt,ge) | c.lt.d $f2, $f4 | si ($f2 < $f4 ) cond =1; si no cond = 0 |