

Números y aritmética

Arquitectura de Computadores – IIC2343

Cuando se trata de números, las personas pensamos en base 10 (números decimales):

- diez símbolos diferentes —0, 1, 2, 3, 4, 5, 6, 7, 8, 9— llamados dígitos decimales, o simplemente, dígitos

... y los representamos muy convenientemente en lo que llamamos la *notación posicional*:

- el verdadero valor de un dígito en un número de varios dígitos depende no sólo del dígito mismo
 - ... sino también de su posición dentro del número

P.ej., cuando escribimos 421, en realidad estamos hablando del número (o valor numérico) que es el resultado de la siguiente operación:

$$4 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

Pero los números pueden ser representados en cualquier base, usando la misma notación posicional

... en particular, en **base 2** (*números binarios*):

- dos símbolos diferentes —0, 1— llamados dígitos binarios o *bits*

P.ej., el número 421 en base 2 se representa así:

1 1 0 1 0 0 1 0 1

... ya que (ver diap. #5):

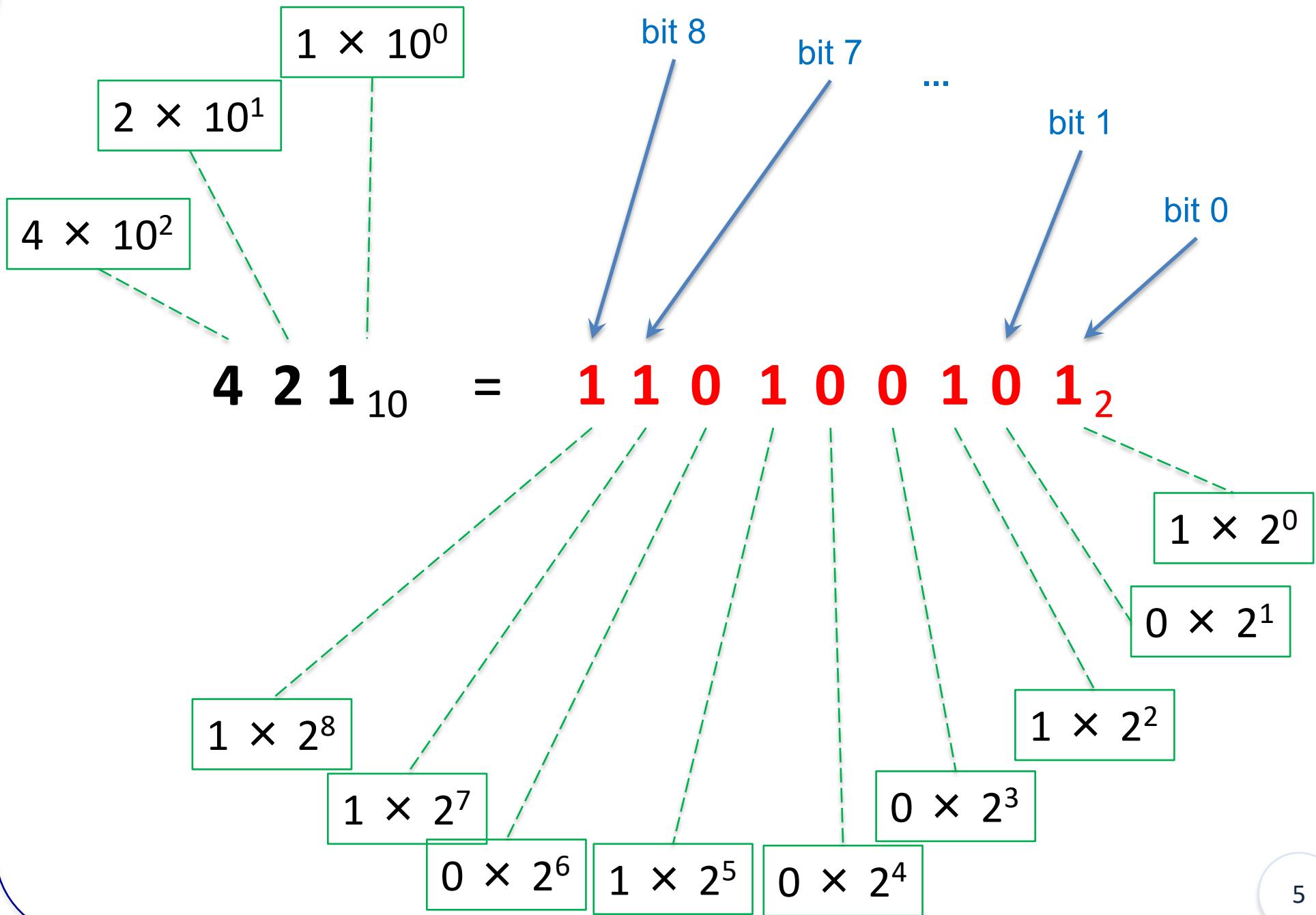
$$\begin{aligned}1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 \\+ 0 \times 2^1 + 1 \times 2^0 = 421\end{aligned}$$

Los bits son los “átomos” de la computación:

- toda información en formato “computacional” se compone de bits

Para poder referirnos a ellos de una manera precisa, numeramos los bits de un número binario 0, 1, 2, 3, ... de derecha a izquierda:

- es decir, desde el bit menos significativo —el que va multiplicado por 2^0 —
- ... al bit más significativo —el que va multiplicado por 2^{n-1} , si el número de bits es n



Tanto en base 10 como en base 2, un número (un valor numérico) tiene una única representación

... es decir, si cambiamos cualquier dígito de 421, el nuevo número va a tener un valor numérico distinto de 421

... y lo mismo con 1 1 0 1 0 0 1 0 1: si cambiamos cualquiera de los 1's por 0's o cualquiera de los 0's por 1's, el valor numérico del número binario resultante va a ser distinto de 421

La cantidad de bits disponible para representar un número queda fija al momento de diseñar el computador:

números de precisión finita

Y esto tiene consecuencias

(Esta no es una limitación de la base 2, sino de que la memoria al interior del computador —independientemente de la tecnología que se use para implementarla— es finita)

P.ej., consideremos el conjunto de números enteros positivos representables mediante tres dígitos decimales, sin punto decimal ni signo:

000, 001, 002, ..., 999

En este caso, es imposible representar ciertos números:

- mayores que 999, negativos, fracciones, irracionales, complejos

Además, el conjunto no es cerrado con respecto a las operaciones aritméticas básicas:

- | | |
|--|---|
| • $600 + 600 = 1200 \rightarrow$ muy grande | $003 - 005 = -2 \rightarrow$ negativo |
| • $050 \times 050 = 2500 \rightarrow$ muy grande | $007 / 002 = 3.5 \rightarrow$ no es un entero |

Finalmente, el álgebra de los números de precisión finita es diferente del álgebra “normal”:

- p.ej., la ley de asociatividad $a + (b - c) = (a + b) - c$ no se cumple si $a = 700$, $b = 400$ y $c = 300$, porque al calcular $a + b$ en el lado derecho produce *overflow*

Estas mismas limitaciones o problemas se dan al representar números enteros positivos (en base 2) mediante, p.ej., 32 bits —típico en un computador moderno

Obviamente, no es que los computadores sean inadecuados para hacer aritmética, sino que

... es importante entender cómo funcionan (que es lo que vamos a ver a continuación)

(En las próximas diapos, vamos a emplear el subíndice 10 cuando escribamos números en base 10, y el subíndice 2 cuando escribamos números en base 2)

En computadores en que las palabras tienen 32 bits, podemos representar 2^{32} patrones diferentes de bits

... los números desde el 0 hasta el $2^{32} - 1 = 4,294,967,295_{10}$

Así, con 32 bits, el valor del número representado como

$$x_{31}x_{30}\dots x_1x_0$$

... es

$$(x_{31} \times 2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Estos números positivos se llaman **números sin signo** (*unsigned numbers*)

Pero también tenemos que representar números negativos

P.ej., podríamos agregar un signo, representado en un bit:

... esta representación se llama ***signo y magnitud***

Problemas:

- el bit de signo, ¿es el de más a la derecha o el de más a la izquierda?
- al sumar, se necesita un paso adicional para saber el valor de este bit
- hay dos ceros —uno positivo y otro negativo

Hay otras opciones, todas con sus pros y sus contras

No habiendo una mejor opción obvia

... los arquitectos computacionales eligieron finalmente la representación llamada ***complemento de 2*** que hacía más simple el hardware:

- 0s a la izquierda significa positivo
- 1s a la izquierda significa negativo

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$$

...

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2,147,483,645_{10}$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2,147,483,646_{10}$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2,147,483,647_{10}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2,147,483,648_{10}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2,147,483,647_{10}$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2,147,483,646_{10}$$

...

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$$

En 32 bits (diap. anterior):

- la mitad positiva, de 0 a $2,147,483,647_{10}$ ($= 2^{31} - 1$) usa la misma representación que antes
- el patrón de bits que sigue ($1000\dots0000_2$) representa el número más negativo, $-2,147,483,648_{10}$ ($= -2^{31}$)
... el cual no tiene un número positivo correspondiente
- y luego viene una secuencia de números negativos de magnitud decreciente, desde $-2,147,483,647_{10}$ ($= 1000\dots001_2$) hasta -1_{10} ($= 1111\dots1111_2$)

Todo computador hoy en día, y desde 1965, usa complemento de 2 para representar **números con signo** (*signed numbers*)

string	unsigned	sign & magnitude	1's complement	2's complement
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

Todos los números negativos tienen un 1 en el bit más significativo:

- *bit de signo* (aunque no es un bit que pueda realmente separarse de la magnitud del número, como vamos a ver)
- basta examinar este bit para saber si un número es positivo o negativo (0 se considera positivo)

Así, con 32 bits en complemento de 2, el valor del número representado como

$$x_{31}x_{30}\dots x_1x_0$$

... es

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Notar la diferencia en la interpretación del valor del bit 31 con respecto al caso de los números sin signo (diap. #10)

En complemento de 2, *overflow* ocurre cuando el resultado de la operación produce un bit de signo incorrecto:

- un 0 (en el bit de más a la izquierda) cuando el número es negativo
- un 1 (en el bit de más a la izquierda) cuando el número es positivo

¿ Cómo determinamos el inverso aditivo de un número binario Y de n bits en complemento de 2 ?

Si miramos la diap. #13 (o la última columna de la diap. #15), vemos que $Y + \bar{Y}$ (invertimos cada bit de Y) = $111\dots111_2 = -1$

... es decir, $Y + \bar{Y} = -1 \Rightarrow -Y = \bar{Y} + 1$

Por lo tanto, el inverso aditivo $-Y$ se obtiene así:

primero, invertimos cada bit de Y ($0 \rightarrow 1, 1 \rightarrow 0$), lo que nos da \bar{Y}

y luego, sumamos 1 al resultado

(por otra parte, si no tomamos en cuenta el signo, $Y + \bar{Y} = 111\dots111_2 = 2^n - 1 \Rightarrow Y + \bar{Y} + 1 = 2^n \Rightarrow Y + (-Y) = 2^n$; de aquí el nombre “complemento de 2”)

¿Qué sabemos?

Números enteros con signo:

Los representamos en *complemento de 2*

Suma:

Los números son sumados bit a bit de derecha a izquierda —lo mismo que en el caso de base 10

... y la reserva (*carry*) se va pasando al próximo par de bits a la izquierda

Resta:

Hace uso de la suma,

... sólo que el sustraendo es convertido a su inverso aditivo antes de ser sumado al minuendo

Overflow es cuando el resultado de una operación no puede ser representado con el número de bits disponible:

- p.ej., si en 32 bits con complemento de 2, el resultado de una operación fuera un número positivo mayor que $2^{31}-1$ (éste es el positivo más grande)
... o un número negativo de magnitud mayor que 2^{31} (-2^{31} es el negativo más grande)

...

...

No puede ocurrir *overflow* cuando sumamos operandos con diferentes signos —uno positivo y el otro negativo:

- ya que la suma no puede ser más grande que uno de los operandos

... ni cuando restamos operandos con el mismo signo —ambos positivos o ambos negativos:

- ya que primero convertimos el sustraendo a su inverso aditivo (es decir, le cambiamos el signo) y luego sumamos

...

...

Cuando ocurre *overflow*, es porque falta un bit para representar el resultado

... entonces el bit de signo —el bit más significativo o más a la izquierda— recibe el valor del resultado

... en lugar de recibir el signo correspondiente al resultado:

- si sumamos dos números positivos y el resultado es negativo
... o viceversa

Es responsabilidad del lenguaje de programación, del sistema operativo y del programa determinar qué hacer en caso de *overflow*:

- el hardware no tiene cómo saber qué conviene hacer en este caso

Multiplicación

Los operandos se llaman *multiplicando* y *multiplicador*

... y el resultado, *producto*

$$\begin{array}{r} 1 \quad 2 \quad 3 \quad \times \quad 4 \quad 5 \\ 6 \quad 1 \quad 5 \\ + \quad 4 \quad 9 \quad 2 \\ = \quad 5 \quad 5 \quad 3 \quad 5 \end{array}$$

— producto intermedio de 123×5
— producto intermedio de 123×4
— suma de los productos intermedios

Algoritmo del colegio (versión informal, ver ej. en diap. anterior):

Tomamos los dígitos del multiplicador uno a uno de derecha a izquierda —es decir, desde el dígito menos significativo al dígito más significativo

... para cada dígito del multiplicador, multiplicamos el multiplicando por ese dígito (\rightarrow *producto intermedio*)

... y escribimos este producto intermedio, desplazado un dígito a la izquierda con respecto al producto intermedio anterior

Finalmente, sumamos los productos intermedios, respetando el desplazamiento de cada uno

Si restringimos los dígitos a 0 y 1 (como es siempre el caso con los números binarios),

... cada paso de la multiplicación es simple

$$\begin{array}{r} 1 & 0 & 1 & 0 \\ \times & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 0 & - = 1010 \times 1 \\ 0 & 0 & 0 & 0 & - = 1010 \times 0 \\ 0 & 0 & 0 & 0 & - = 1010 \times 0 \\ + & 1 & 0 & 1 & 0 & - = 1010 \times 1 \\ \hline = & 1 & 0 & 1 & 1 & 0 & 1 & 0 & -\text{suma} \end{array}$$

Algoritmo de multiplicación binaria (versión informal, ver ej. en diap. anterior):

Colocar una copia del multiplicando en el lugar correcto*, si el dígito del multiplicador es 1 (= 1 × multiplicando)

... o bien colocar 0 en el lugar correcto*, si el dígito del multiplicador es 0 (= 0 × multiplicando)

* en cada nueva iteración, se escribe desplazándolo un dígito a la izquierda

Hardware necesario para multiplicación

El número de dígitos en el producto es mayor (¿cuánto mayor?) que el número de dígitos en cualquiera de los operandos → posibilidad de overflow

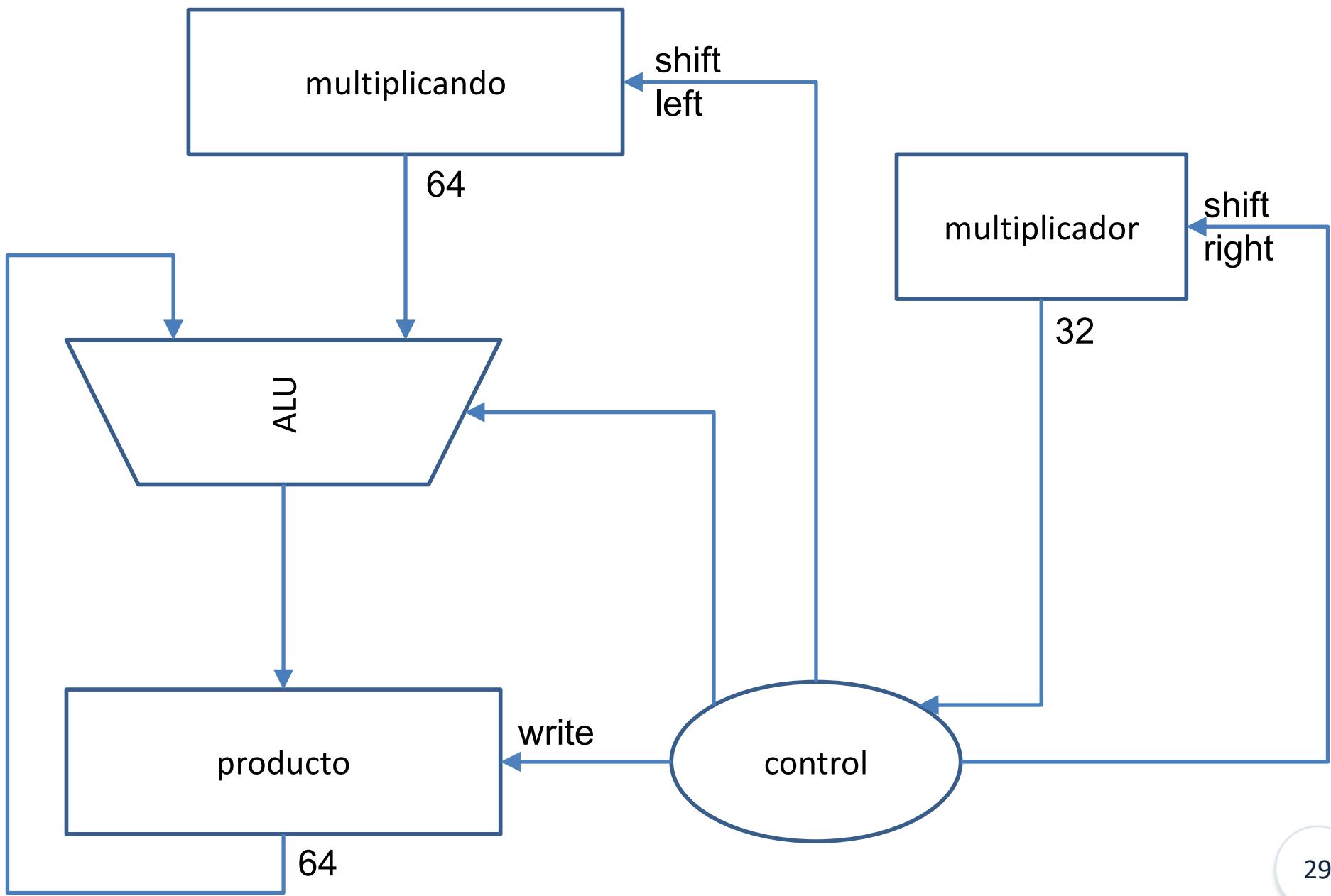
Necesitamos registros para

... el multiplicando (con capacidad de *shift* a la izquierda, ¿con cuántos bits?)

... el multiplicador (con capacidad de *shift* a la derecha, ¿por qué?) y

... el producto (con el doble de bits que los operandos)

Hardware para multiplicación (versión secuencial)



Algoritmo de multiplicación de números de 32 bits sin signo, usando el hardware anterior

Partir

1. Examinar el dígito de más a la derecha del multiplicador
 - 1a. Si es 1, sumar el multiplicando al producto y poner el resultado en el registro del producto
2. Desplazar (*shift*) el registro del multiplicando a la izquierda un bit
3. Desplazar (*shift*) el registro del multiplicador a la derecha un bit
4. Verificar si los pasos 1, 2 y 3 se han repetido 32 veces
 - 4a. Si aún faltan repeticiones, ir a 1

Terminar