Universidad Nacional de Ingeniería
Facultad de Ciencias

Introducción a la Ciencia de la Computación

Operaciones en Datos

Prof: J. Solano
2011-I

Objetivos

Despues de estudiar este cap. el estudiante sera capaz de:

- **Listar las tres categorías de operaciones que se realizan en los datos.**
- **Realizar operaciones lógicas binarias y unitarias en los patrones de bits.**
- **Distinguir entre las operaciones de cambio (shift) lógicas y aritméticas.**
- **Realizar sumas y restas de números enteros cuando están almacenados en formato de complemento a dos.**
- **Realizar sumas y restas de números enteros cuando se guardan en formato signo y magnitud.**
- **Realizar operaciones de suma y resta en reales almacenados en formato de punto flotante.**

OPERACIONES LÓGICAS

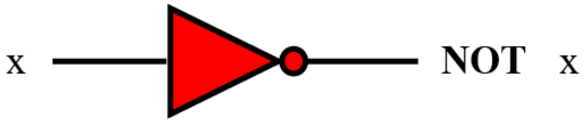
En el capítulo anterior vimos que los datos son almacenados en el computador como patrones de bits. Operaciones logícas se refieren a las operaciones que aplican la misma operación básica sobre bits individuales de un patrón, o en dos bits correspondientes en dos patrones. Esto significa que podemos definir las operaciones lógicas a nivel de bits (bit level) y a nivel de patrones (pattern level).

Una operación lógica a nivel de patrones son *n* operaciones lógicas, del mismo tipo, a nivel de bits, donde *n* es el número de bits en el patrón.

Operaciones lógicas a nivel de bits

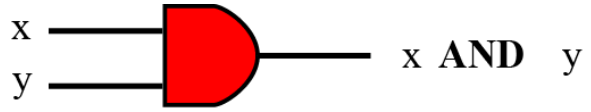
Un bit puede tomar dos valores: 0 ó 1. Solo interpretamos el 0 como el valor *false* (falso) y 1 como el valor *true* (verdadero), y podemos aplicar las operaciones definidas en el algebra booleana para manipular bits. **Boolean algebra** (algebra boolena), llamada así en honor de George Boole, se convirtió en un campo especial de la Matemática llamado Lógica. Una gran aplicación del algebra booleana está en la construcción de circuitos lógicos en computadores. En esta sección solo discutiremos cuatro operaciones a nivel de bits, que son usadas para manipular bits: **NOT**, **AND**, **OR**, y **XOR**.

Algebra Booleana es la base de los circuitos lógicos de computadores y muchos dispositivos electrónicos



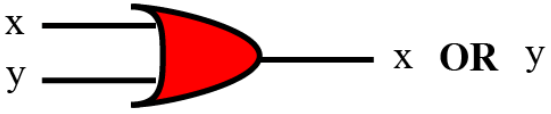
NOT

x	NOT x
0	1
1	0



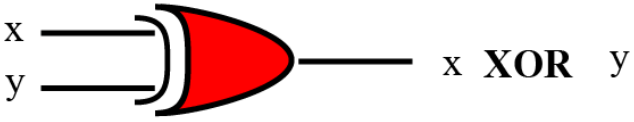
AND

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1



OR

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1



XOR

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Operaciones lógicas a nivel de bits

NOT

El operador NOT es un operador unitario: toma solo un input. El bit ouput es el complemento del input.

AND

El operador AND es un operador binario: toma dos inputs. El bit ouput es 1 si ambos inputs son 1s y 0 en los otros tres casos.

Para x = 0 o 1 $\quad x$ AND 0 $\rightarrow$ 0 $\qquad$ 0 AND x $\rightarrow$ 0

OR

El operador OR es un operador binario: toma dos inputs. El bit ouput es 0 si ambos inputs son 0s y es 1 en los otros tres casos.

For *x* = 0 or 1 *x* OR 1 → 1 1 OR *x* → 1

XOR

El operador XOR es un operador binario como el operador OR con solo una diferencia: el bit ouput es 0 si ambos inputs son 1s.

For *x* = 0 or 1
1 XOR *x* → NOT *x* *x* XOR 1 → NOT *x*

Ejemplo 1

En inglés se usa la conjunción “or” a veces para referirse a inclusive-or, y otras veces para exclusive-or.

a. The sentence “I would like to have a car or a house” uses “or” in the inclusive sense—I would like to have a car, a house or both.

b. The sentence “Today is either Monday or Tuesday” uses “or” in the exclusive sense—today is either Monday or Tuesday, but it cannot be both.

Ejemplo 2

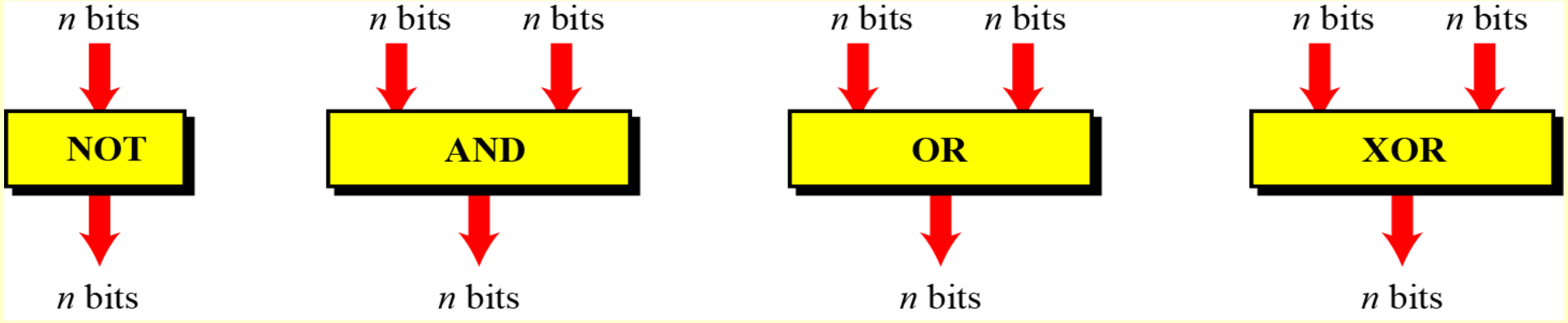
El operador XOR no es un operador nuevo. Siempre podemos simularlo usando los otros tres operadores. Las siguientes dos operaciones son equivalentes

$$x \text{ XOR } y \leftrightarrow [x \text{ AND } (\text{NOT } y)] \text{ OR } [(\text{NOT } x) \text{ AND } y]$$

La equivalencia puede probarse si hacemos la tabla de verdad para ambos

Operaciones lógicas a nivel de patrones

Los mismos cuatro operadores (NOT, AND, OR, y XOR) pueden ser aplicados a un patrón de n-bits. El efecto es el mismo de aplicar cada operador a cada bit individual para NOT y para cada par de bits correspondientes para los otros tres operadores. Aqui se muestra los cuatro operadores con patrones de input y output.



Operadores lógicos aplicados a patrones de bits

Ejemplo 3

Usar el operador NOT en el patrón de bits 10011000

Solución

La solución es mostrada abajo. Notar que el operador NOT cambia cada 0 a 1 y cada 1 a 0.

NOT	1	0	0	1	1	0	0	0	Input
	0	1	1	0	0	1	1	1	Output

Ejemplo 4

Usar el operador AND en los patrones de bits 10011000 y 00101010

Solución

La solución es mostrada abajo. Notar que solo un bit en el output es 1, cuando los dos bits correspondientes son 1s

	1	0	0	1	1	0	0	0	Input 1
AND	0	0	1	0	1	0	1	0	Input 2
	0	0	0	0	1	0	0	0	Output

Ejemplo 5

Usar el operador OR en los patrones de bits 10011001 y 00101110

Solución

La solución es mostrada abajo. Notar que solo un bit en el output es 0, cuando los dos bits correspondientes son 0s

	1	0	0	1	1	0	0	1	Input 1
OR	0	0	1	0	1	1	1	0	Input 2
	1	0	1	1	1	1	1	1	Output

Ejemplo 6

Usar el operador XOR en los patrones de bits 10011001 y 00101110

Solución

La solución es mostrada abajo. Comparar el output con el del ejemplo anterior. La única diferencia es que cuando los dos bits inputs son 1s, el resultado es 0 (el efecto de exclusión).

	1	0	0	1	1	0	0	1	Input 1
XOR	0	0	1	0	1	1	1	0	Input 2
	1	0	1	1	0	1	1	1	Output

Aplicaciones

Las cuatro operaciones lógicas pueden ser usadas para modificar un patrón de bits.

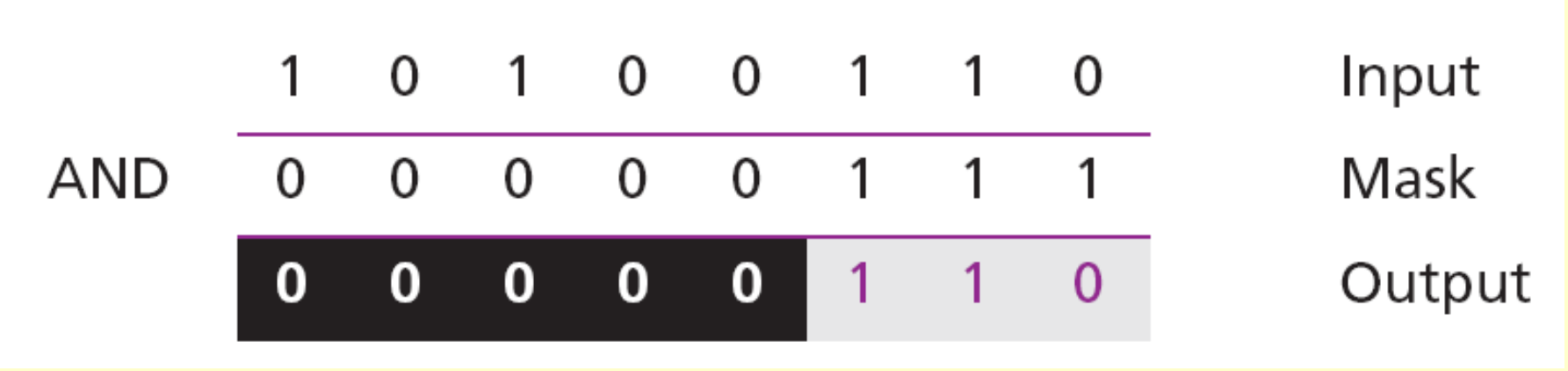
- **Complementing (NOT)**
- **Unsetting (desajuste) (AND)**
- **Setting (ajuste) (OR)**
- **Flipping (voltear) (XOR)**

Ejemplo 7

Usar una máscara (mask) para desajustar (clear) los cinco bits mas a la izquierda de un patrón de bits. Probar la máscara con un patrón 10100110.

Solución

La máscara es 00000111. El resultado de aplicar la máscara es:

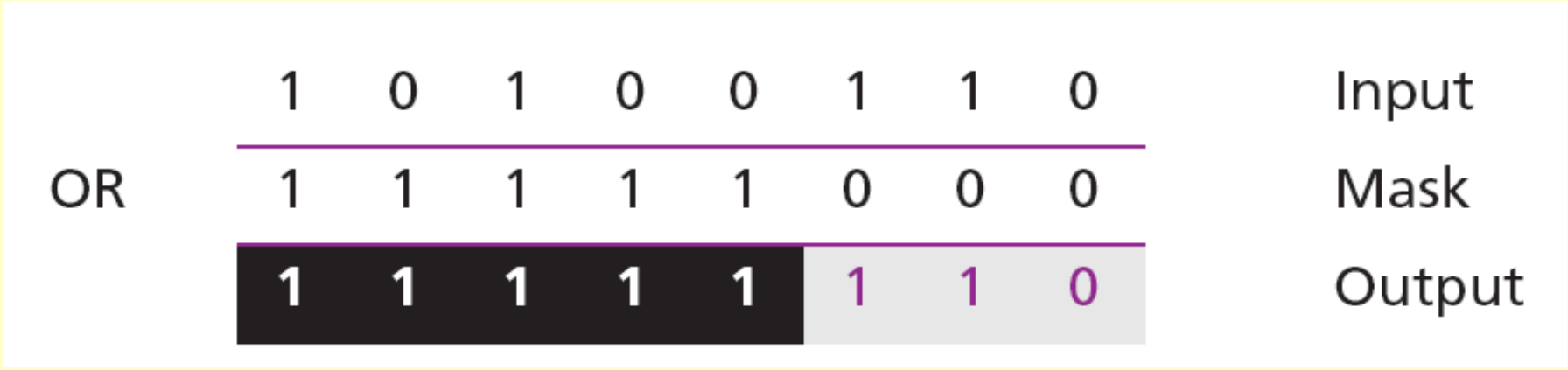


Ejemplo 8

Usar una máscara para ajustar los cinco bits mas a la izquierda de un patrón de bits. Probar la máscara con un patrón 10100110.

Solución

La máscara es 11111000. El resultado de aplicar la máscara es:



Ejemplo 9

Usar una máscara para voltear (flip) los cinco bits mas a la izquierda de un patrón de bits. Probar la máscara con un patrón 10100110.

Solución

La máscara es 11111000. El resultado de aplicar la máscara es:

	1	0	1	0	0	1	1	0	Input 1
XOR	1	1	1	1	1	0	0	0	Mask
	0	**1**	**0**	**1**	**1**	1	1	0	Output

OPERACIONES SHIFT (DESPLAZAMIENTO)

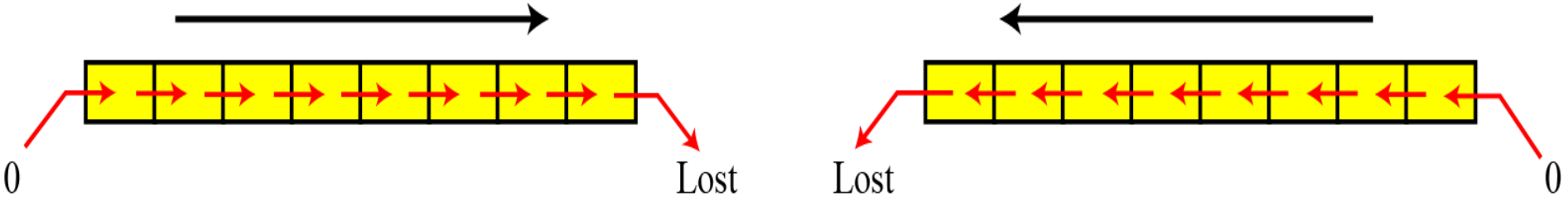
Operaciones Shift mueve los bits en un patrón, cambiando la posición de los bits. Ellos pueden mover bits a la izquierda o a la derecha. Podemos dividir operaciones shift en dos categorías: **operaciones shift lógicas** y **operaciones shift aritméticas**.

Operaciones shift lógicas

Una operación shift lógica se aplica a un patrón que no representa un número con signo. La razón es que estas operaciones shift pueden cambiar el signo del número que se define por el bit más a la izquierda en el patrón. Se distinguen dos tipos de operaciones shift lógicas, como se describe a continuación:

- **Shift lógico**

- **Shift circular lógico (Rotar)**



a. Logical right shift

b. Logical left shift

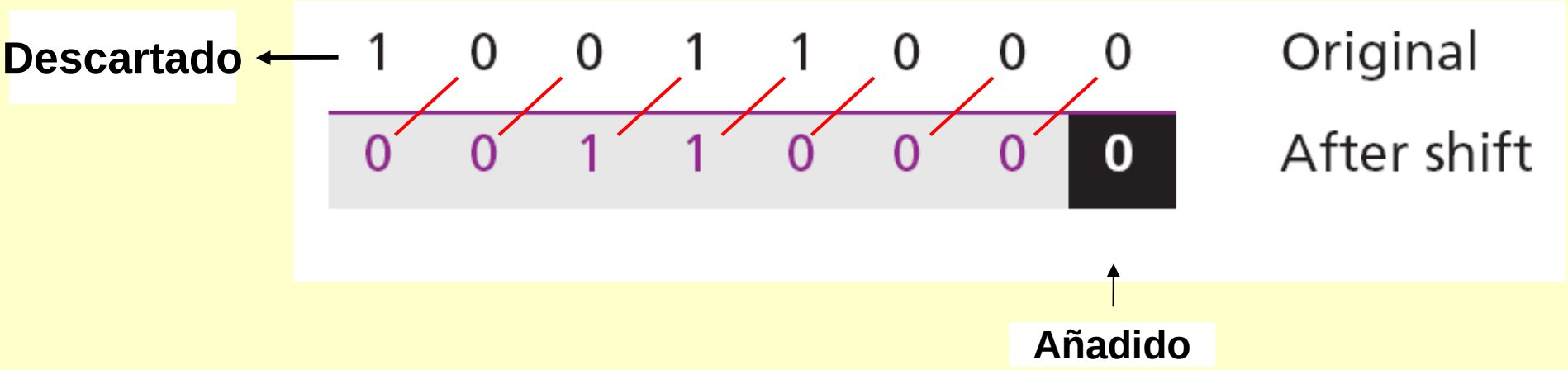
Operaciones shift lógicas

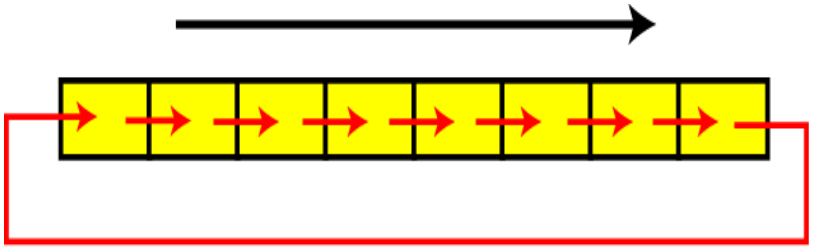
Ejemplo 10

Usar una operación shift lógica izquierda en el patrón de bits 10011000.

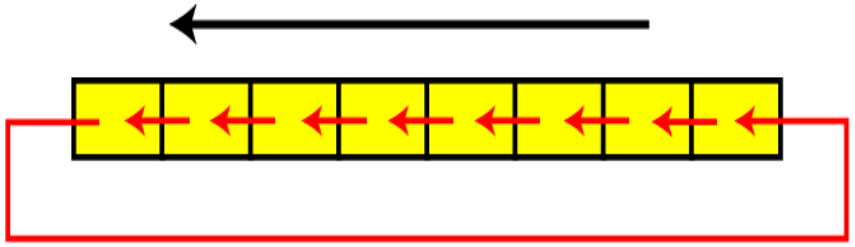
Solución

La solución es mostrada abajo. El bit mas a la izquierda se pierde y un 0 es insertado en el bit mas a la derecha





a. Circular right shift



b. Circular left shift

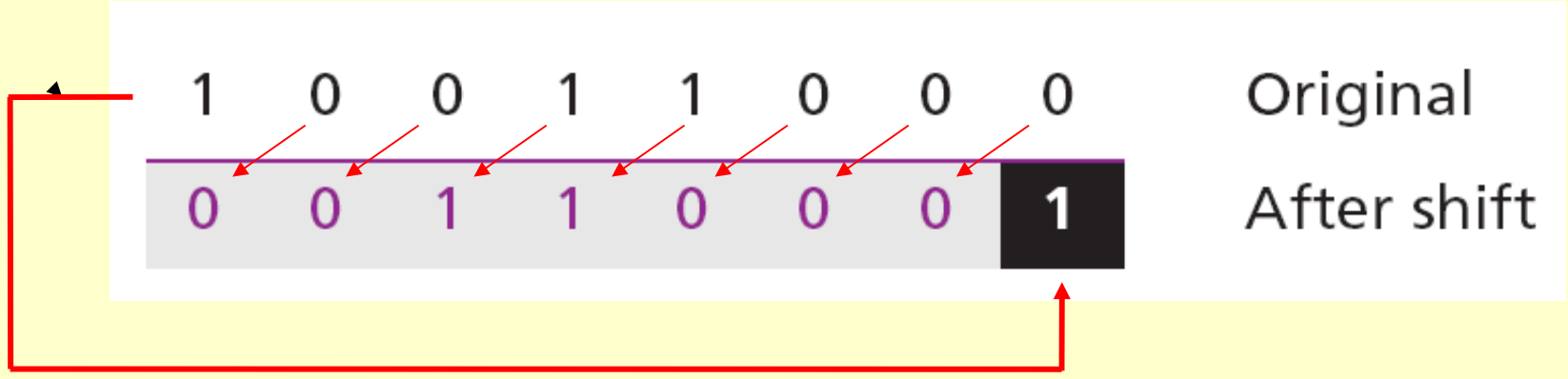
Operaciones shift circulares

Ejemplo 11

Usar una operación shift circular izquierda en el patrón de bits 10011000.

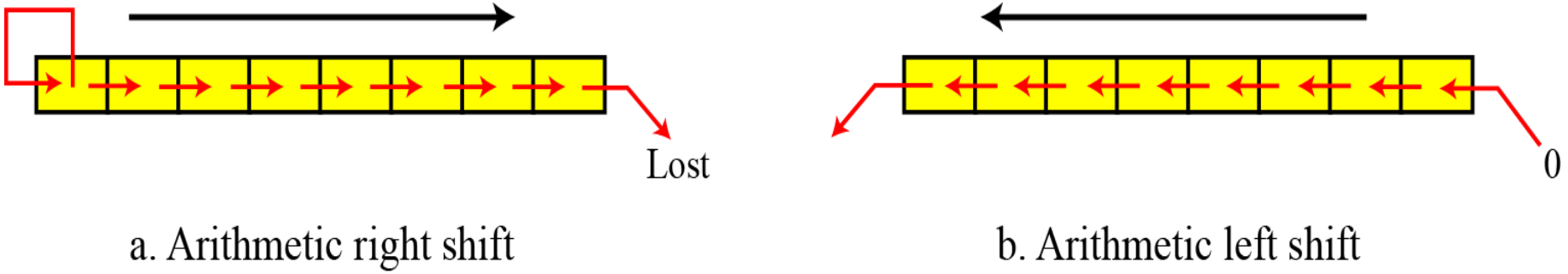
Solución

La solución es mostrada abajo. El bit mas a la izquierda es circulado y se convierte en el bit mas a la derecha.



Operaciones shift aritméticas

Las operaciones shift aritméticas asumen que el patrón de bits es un entero con signo en formato complemento a dos. Shift aritmético derecho se utiliza para dividir un entero en dos, mientras que el shift aritmético izquierdo se usa para multiplicar un número entero por dos.



a. Arithmetic right shift

b. Arithmetic left shift

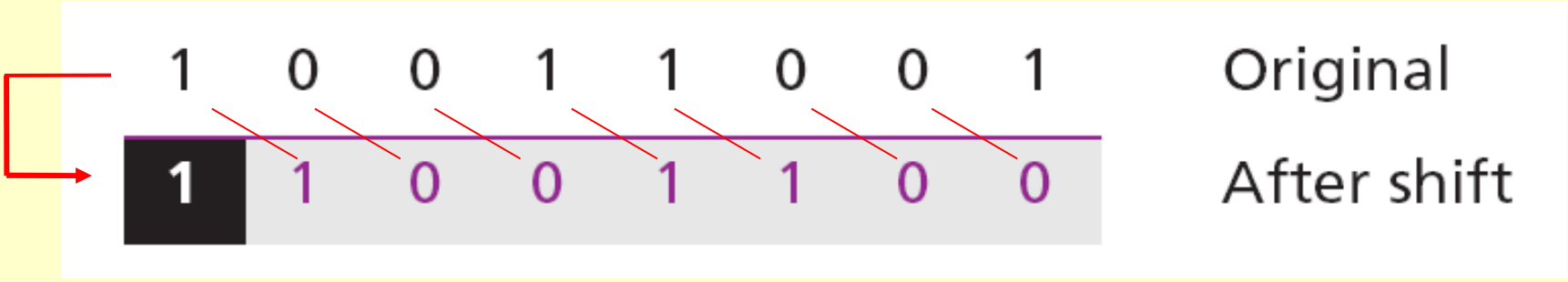
Operaciones shift aritméticas

Ejemplo 12

Usar una operacion shift aritmética derecha en el patrón de bits 10011001. El patrón es un entero en formato complemento dos

Solución

La solución es mostrada abajo. El bit mas a la izquierda es retenido y también copiado a su bit vecino a su derecha



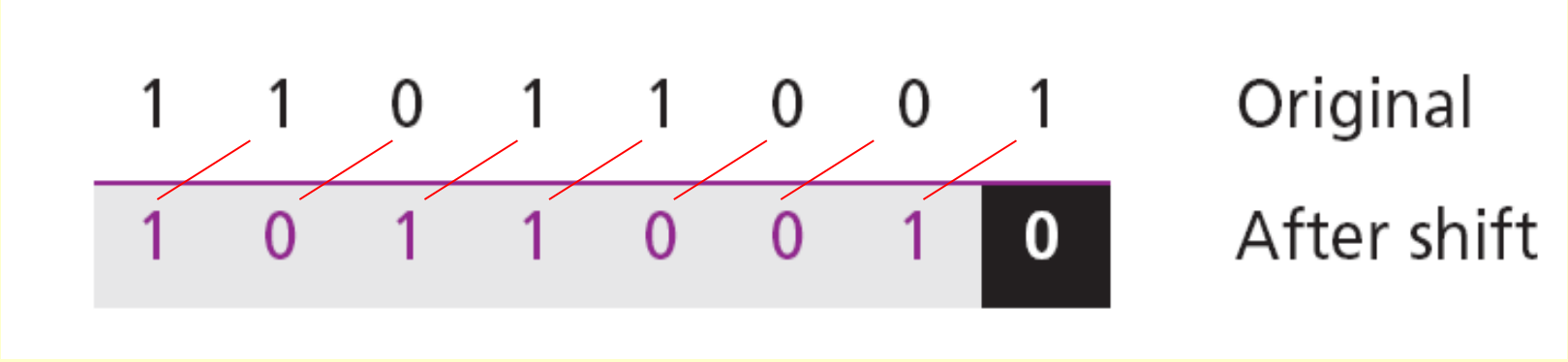
El número original era -103 y el nuevo número es -52, que es el resultado de dividir -103 por 2 truncado al menor entero.

Ejemplo 13

Usar una operacion shift aritmética izquierda en el patrón de bits 11011001. El patrón es un entero en formato complemento dos

Solución

La solución es mostrada abajo. El bit mas a la izquierda es perdido y un 0 es insertado en el bit mas a la derecha



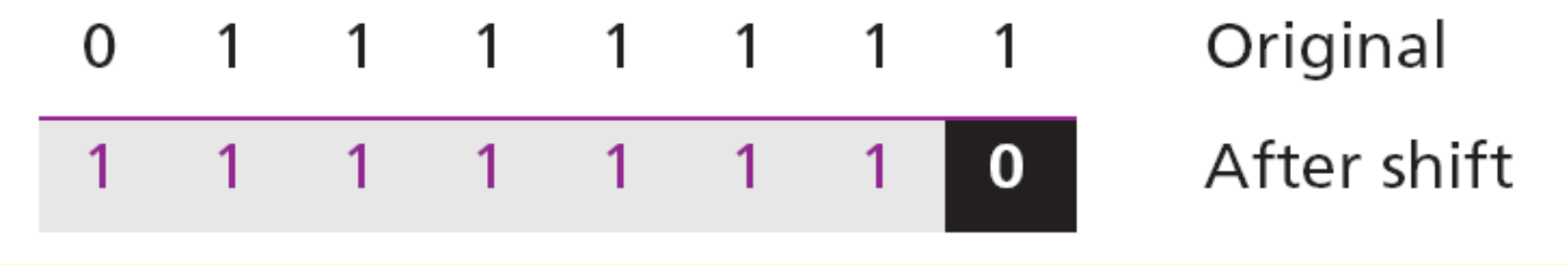
El número original era -39 y el nuevo número es -78. El número original es multiplicado por dos. La operación es valida porque no ocurrió underflow.

Ejemplo 14

Usar una operación shift aritmética izquierda en el patrón de bits 01111111. El patrón es un entero en formato complemento a dos.

Solución

La solución es mostrada abajo. El bit mas a la izquierda es perdido y un 0 es insertado en el bit mas a la derecha



El número original era 127 y el nuevo número es -2. Aquí el resultado no es válido porque ocurrió un overflow. La respuesta esperada 127 x 2 = 254 no puede ser representada por un patrón de 8 bits, en formato complemento a dos.

Ejemplo 15

Combinando operaciones lógicas y operaciones shift lógicas nos da algunas herramientas para manipular patrones de bits. Supongamos que tenemos un patrón y necesitamos utilizar el tercer bit (de la derecha) de este patrón en un proceso de toma de decisiones. Queremos saber si este bit particular es 0 o 1. A continuación se muestra cómo podemos hallarlo.

	h	g	f	e	d	c	b	a	
	0	h	g	f	e	d	c	b	One right shift
	0	0	h	g	f	e	d	c	Two right shifts
AND	0	0	0	0	0	0	0	1	Mask
	0	0	0	0	0	0	0	c	Result

Podemos probar el resultado: si es un entero sin signo 1, el bit blanco era 1, mientras que si el resultado es un entero sin signo 0, el bit blanco era 0.

OPERACIONES ARITMÉTICAS

Operaciones aritméticas implican sumar, restar, multiplicar y dividir. Podemos aplicar estas operaciones a los números enteros y a los números de punto flotante.

Operaciones aritméticas en enteros

Todas las operaciones aritméticas como suma, resta, multiplicación y división se puede aplicar a enteros. A pesar que la multiplicación (división) de enteros puede ser implementada usándose suma (resta) repetida, el procedimiento no es eficiente. Hay procedimientos más eficientes para la multiplicación y división, como procedimientos de cabina (**Booth procedures**), pero estos están fuera del alcance de este curso. Por esta razón, sólo discutimos aquí sumas y restas de números enteros.

Enteros en formato complemento a dos

Cuando se encuentra la operación resta, el computador simplemente cambia a una operación de suma, pero cambia a formato complemento a dos el segundo número. En otras palabras:

$$\mathbf{A - B \leftrightarrow A + (\overline{B} + 1)}$$

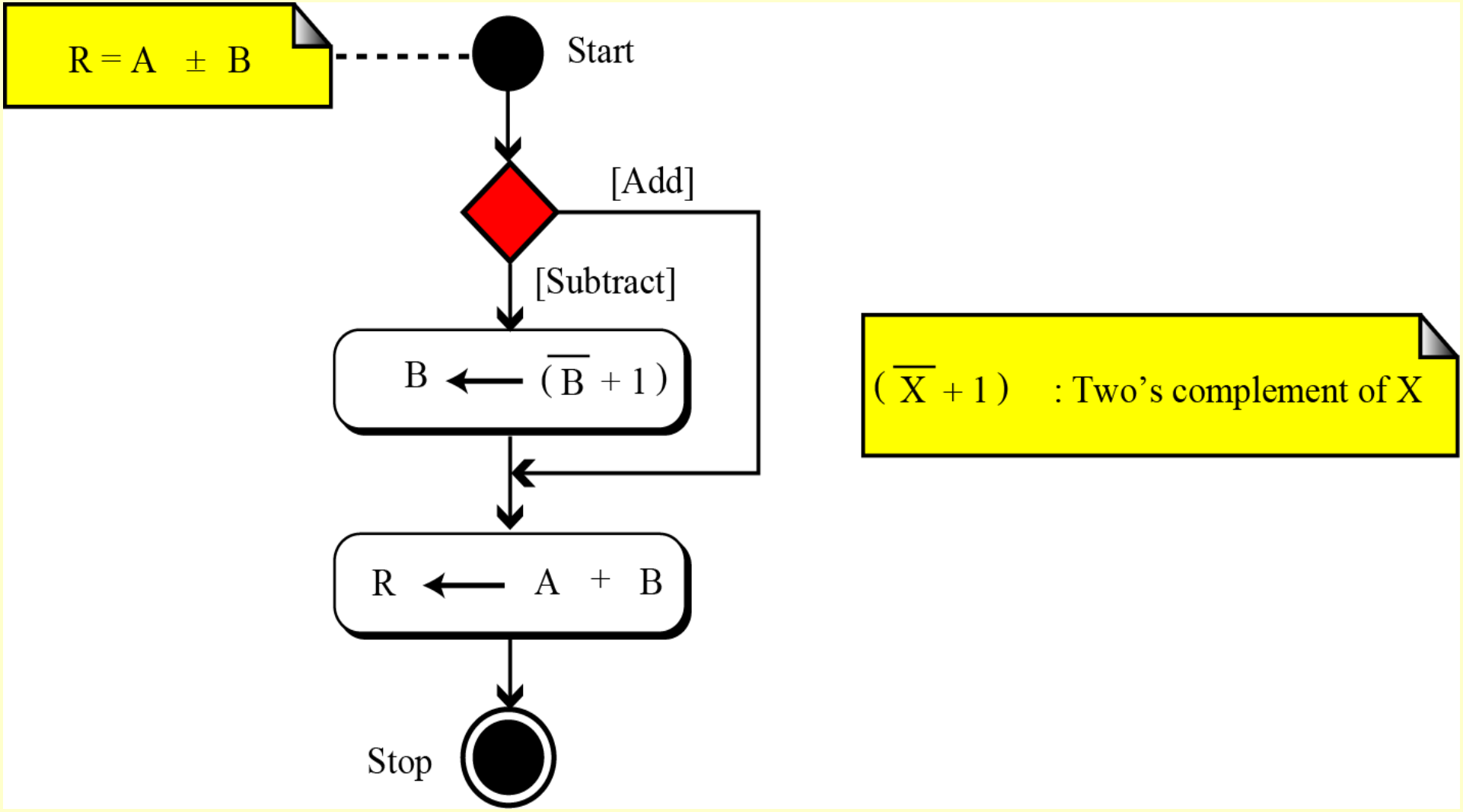
Donde $\overline{B}$ es el complemento a uno de B y

$\overline{(B + 1)}$ es el complemento a dos de B

Debemos recordar que sumamos enteros columna por columna. La siguiente tabla muestra la suma y lo que se lleva (carry) (C).

Table 4.1 Carry and sum resulting from adding two bits

Column	*Carry*	*Sum*	*Column*	*Carry*	*Sum*
Zero 1s	0	0	Two 1s	1	0
One 1	0	1	Three 1s	1	1



Suma y resta de enteros en formato complemento a dos

Ejemplo 16

Dos enteros A y B son almacenados en formato complemento a dos. Mostrar como B es sumado a A

$$\mathbf{A = (00010001)_2} \qquad \mathbf{B = (00010110)_2}$$

Solución

La operación es suma. A es sumado a B y el resultado es almacenado en R. (+17) + (+22) = (+39).

			1						Carry
	0	0	0	1	0	0	0	1	A
+	0	0	0	1	0	1	1	0	B
	0	0	1	0	0	1	1	1	R

Ejemplo 17

Dos enteros A y B son almacenados en formato complemento a dos. Mostrar como B es sumado a A

$$\mathbf{A = (00011000)_2} \qquad \mathbf{B = (11101111)_2}$$

Solución

La operación es suma. A es sumado a B y el resultado es almacenado en R. (+24) + (-17) = (+7).

1	1	1	1	1					Carry
	0	0	0	1	1	0	0	0	A
+	1	1	1	0	1	1	1	1	B
	0	0	0	0	0	1	1	1	R

Ejemplo 18

Dos enteros A y B son almacenados en formato complemento a dos. Mostrar como B es restado de A

$$\mathbf{A = (00011000)_2} \qquad \mathbf{B = (11101111)_2}$$

Solución

La operación es resta. A es sumado a $(\overline{B} + 1)$ y el resultado es almacenado en R. (+24) - (-17) = (+41).

			1						Carry
	0	0	0	1	1	0	0	0	A
+	0	0	0	1	0	0	0	1	$(\overline{B} + 1)$
	0	0	1	0	1	0	0	1	R

Ejemplo 19

Dos enteros A y B son almacenados en formato complemento a dos. Mostrar como B es restado de A

$$\mathbf{A = (11011101)_2} \qquad \mathbf{B = (00010100)_2}$$

Solución

La operación es resta. A es sumado a $(\overline{B} + 1)$ y el resultado es almacenado en R. (-35) - (+20) = (-55).

1	1	1	1	1	1			Carry
	1	1	0	1	1	1	0	1	A
+	1	1	1	0	1	1	0	0	$(\overline{B} + 1)$
	1	1	0	0	1	0	0	1	R

Ejemplo 20

Dos enteros A y B son almacenados en formato complemento a dos. Mostrar como B es sumado a A

$$\mathbf{A = (01111111)_2} \qquad \mathbf{B = (00000011)_2}$$

Solución

La operación es suma. A es sumado a B y el resultado es almacenado en R.

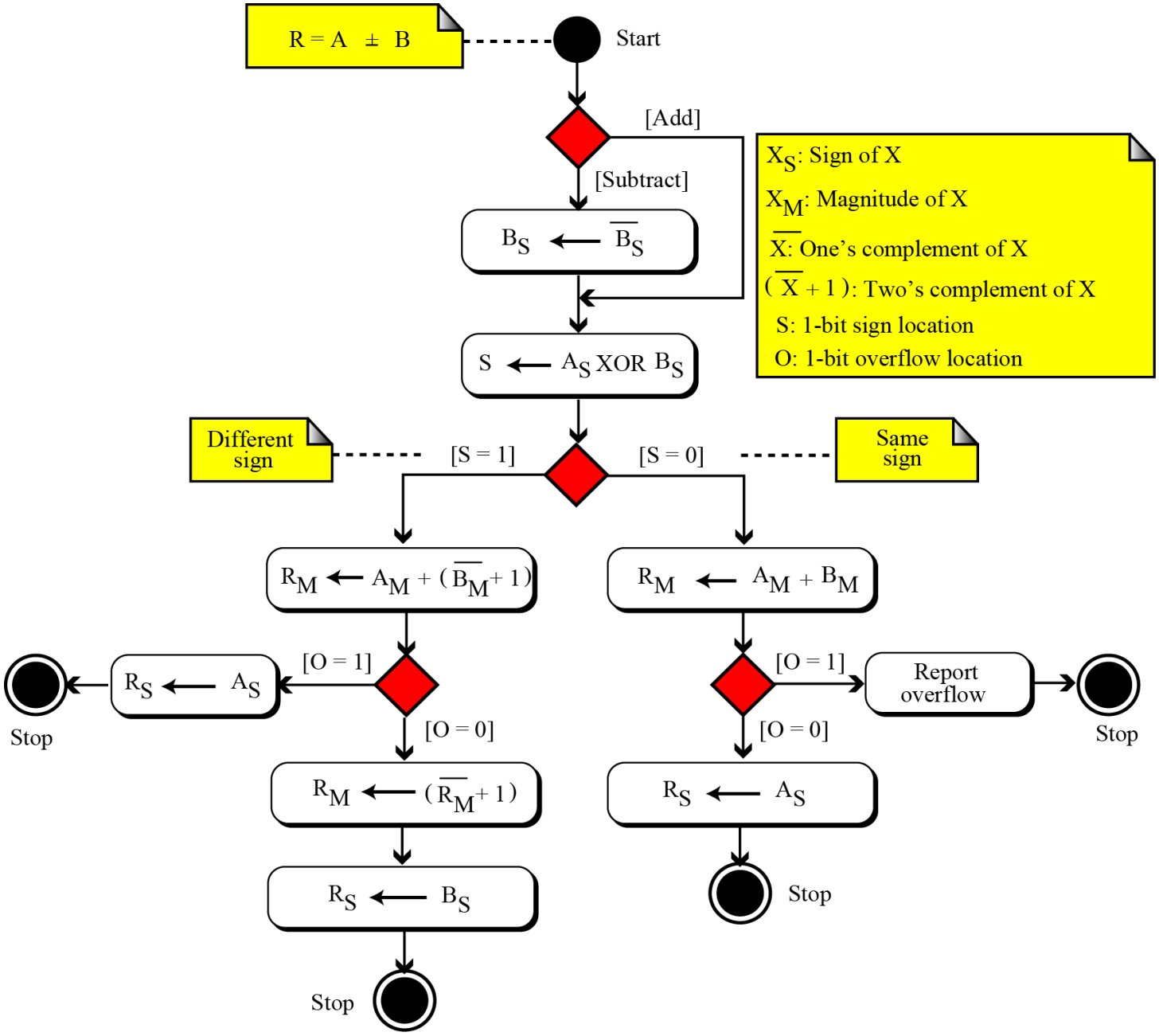
	1	1	1	1	1	1	1	Carry
	0	1	1	1	1	1	1	1 A
+	0	0	0	0	0	0	1	1 B
	1	0	0	0	0	0	1	0 R

Esperabamos el resultado 127 + 3 = 130, pero la respuesta es -126. El error es debido al overflow, porque el valor esperado (+130) no está en el rango -128 a +127.

Cuando hacemos operaciones aritméticas en números en un computador, debemos recordar que cada número y el resultado deben estar en el rango definido por la ubicación de bits en memoria.

Enteros en formato signo y magnitud

Suma y resta para enteros en representación signo y magnitud es un poco compleja. Tenemos cuatro combinaciones de signos (dos signos, para cada uno de dos valores) para suma y cuatro diferentes condiciones para resta. Esto significa que debemos considerar ocho situaciones diferentes.
Sin embargo, si verificamos primero los signos, podemos reducir esos casos.



Sumas y restas de enteros en formato signo y magnitud

Ejemplo 22

Dos enteros A y B son almacenados en formato signo y magnitud. Mostrar como B es sumado a A

$$\mathbf{A = (0\ 0010001)_2} \qquad \mathbf{B = (1\ 0010110)_2}$$

Solución

La operación es suma: el signo de B no se cambia. $S = A_S$ XOR $B_S = 1$; $R_M = A_M + (\overline{B_M} + 1)$. Como no hay overflow, necesitamos tomar el complemento a dos de R_M. El signo de R es el signo de B. (+17) + (-22) = (-5).

No overflow

									Carry
A_S	0		0	0	1	0	0	0	1	A_M
B_S	1	+	1	1	0	1	0	1	0	$\overline{(B_M + 1)}$
			1	1	1	1	0	1	1	R_M
R_S	1		0	0	0	0	1	0	1	$R_M = \overline{(R_M + 1)}$

Ejemplo 23

Dos enteros A y B son almacenados en formato signo y magnitud. Mostrar como B es restado de A

$$\mathbf{A = (1\ 1010001)_2} \qquad \mathbf{B = (1\ 0010110)_2}$$

Solución

La operación es resta: el signo de B se cambia $\mathbf{(S_B = \overline{S_B})}$. $S = A_S$ XOR $B_S = 1$; $R_M = A_M + (\overline{B}_M + 1)$. Como hay overflow, el valor de R_M es el final. El signo de R es el signo de A. (-81) - (-22) = (-59).

		Overflow →	1						Carry
A_S	1			1	0	1	0	0	0	1	A_M
B_S	1		+	1	1	0	1	0	1	0	$(\overline{B}_M + 1)$
R_S	1			0	1	1	1	0	1	1	R_M

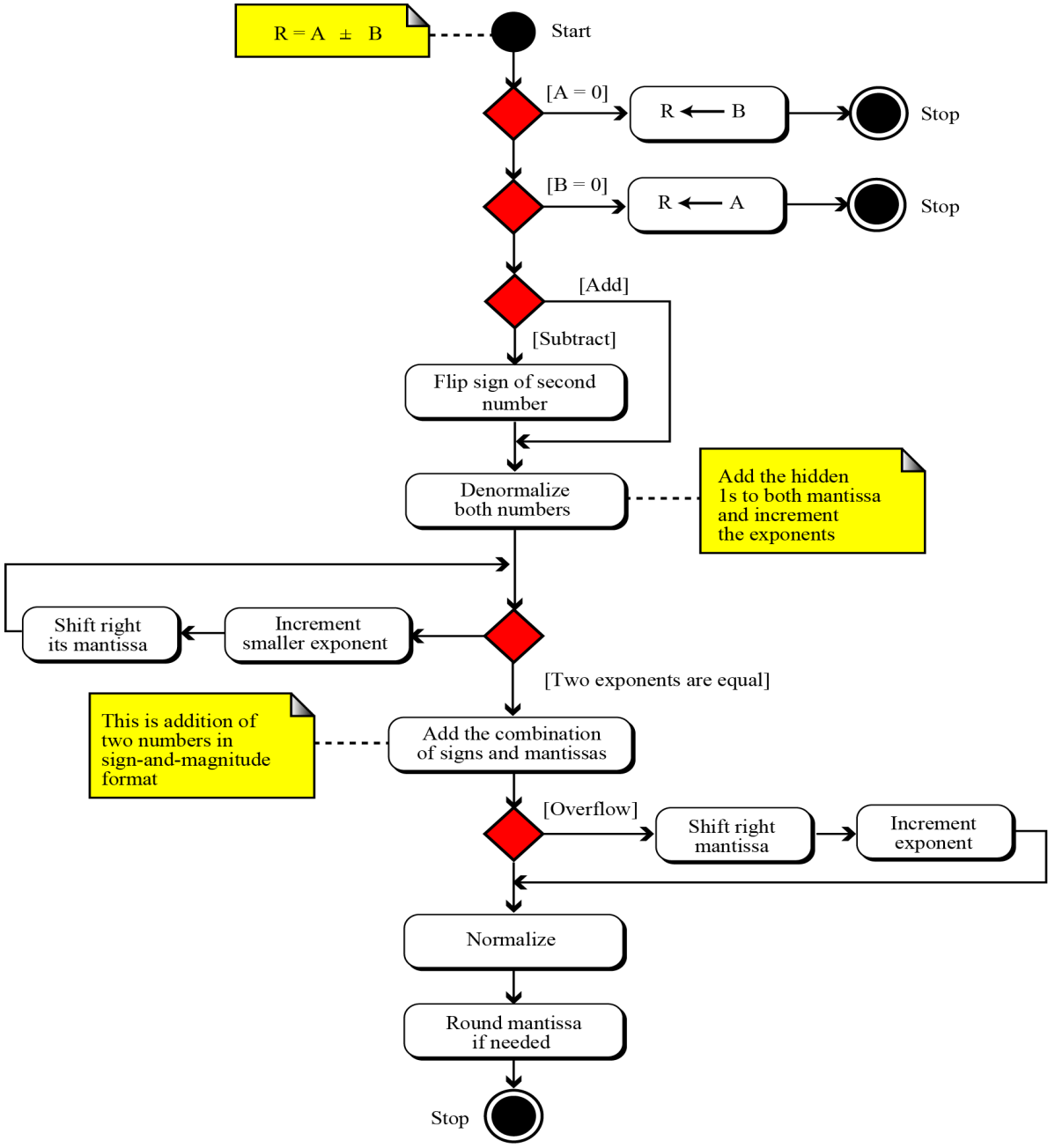
Operaciones aritméticas en reales

Todas las operaciones aritméticas como suma, resta, multiplicación y división se pueden aplicar a reales almacenados en formato punto flotante. Multiplicación de dos reales envuelve multiplicación de dos enteros en representación signo y magnitud. División de dos reales envuelve división de dos enteros en representación signo y magnitud.

Como no discutimos multiplicación ni división de enteros en representación signo y magnitud, no discutiremos aquí multiplicación ni división de reales. Solo mostraremos sumas y restas de reales.

Suma y resta de reales

Suma y resta de números reales almacenados en punto flotante se reduce a suma y resta de dos enteros almacenados en signo y magnitud (combinación de signo y mantisa) después del alineamiento del punto decimal.



Suma y resta de reales en formato punto flotante

Ejemplo 24

Mostrar como el computador halla el resultado (+5.75) + (+161.875) = (+167.625)

Solución

Como vimos en el capítulo de Almacenamiento de Datos, esos dos números son almacenados en formato punto flotante, como mostrado antes, pero necesitamos recordar que cada número tiene un 1 escondido (que no es almacenado pero si asumido).

	S	E	M
A	0	10000001	01110000000000000000000
B	0	10000110	01000011110000000000000

Ejemplo 24 (continuación)

Los primeros pasos del diagrama UML mostrado antes no son necesarios. Primero de-normalizamos los números añadiendo los 1s escondidos a la mantisa e incrementando el exponente. Ahora ambas mantisas de-normalizadas son de 24 bits e incluyen los 1s escondidos. Ellos deberían ser almacenados en una ubicación de memoria de 24 bits. Cada exponente es incrementado.

	S	E	Denormalized M
A	0	10000010	101110000000000000000000
B	0	10000111	101000011110000000000000

Ejemplo 24 (continuación)

Ahora hacemos suma de signo y magnitud, tratando el signo y la mantisa de cada número como un entero almacenado en representación signo y magnitud.

	S	E	Denormalized M
R	0	10000111	10100111101000000000000

No hay overflow en la mantisa, entonces normalizamos.

	S	E	M
R	0	10000110	01001111010000000000000

La mantisa es solo 23 bits, no se necesita redondeo. E = $(10000110)_2$ = 134 M = 0100111101. En otras palabras, el resultado es $(1.0100111101)_2 \times 2^{134-127} = (10100111.101)_2 =$ **167.625**.

Ejemplo 25

Mostrar como el computador halla el resultado (+5.75) + (−7.0234375) = − 1.2734375

Solucion

Esos dos números pueden ser almacenados en formato punto flotante, como se muestra abajo:

	S	E	M
A	0	10000001	01110000000000000000000
B	1	10000001	11000001100000000000000

La de-normalización resulta en:

	S	E	Denormalized M
A	0	10000010	**1**01110000000000000000000
B	1	10000010	**1**11000001100000000000000

Ejemplo 25 (continuación)

Alineamiento no es necesario (ambos tienen el mismo exponente), entonces aplicamos la operación suma a la combinación de signo y mantisa. El resultado se muestra abajo, en donde el signo del resultado es negativo.

	S	E	Denormalized M
R	1	10000010	00101000110000000000000

Ahora necesitamos normalizar. Decrecemos el exponente tres veces y desplazamos (shift) la mantisa de-normalizada a la izquierda tres posiciones:

	S	E	M
R	1	01111111	01000110000000000000000

Ejemplo 25 (continuación)

La mantisa es ahora de 24 bits, entonces redondeamos a 23 bits.

	S	E	M
R	1	01111111	01000110000000000000000

El resultado es $R = -2^{127-127} \times 1.0100011 =$ **-1.2734375**, como se experaba.