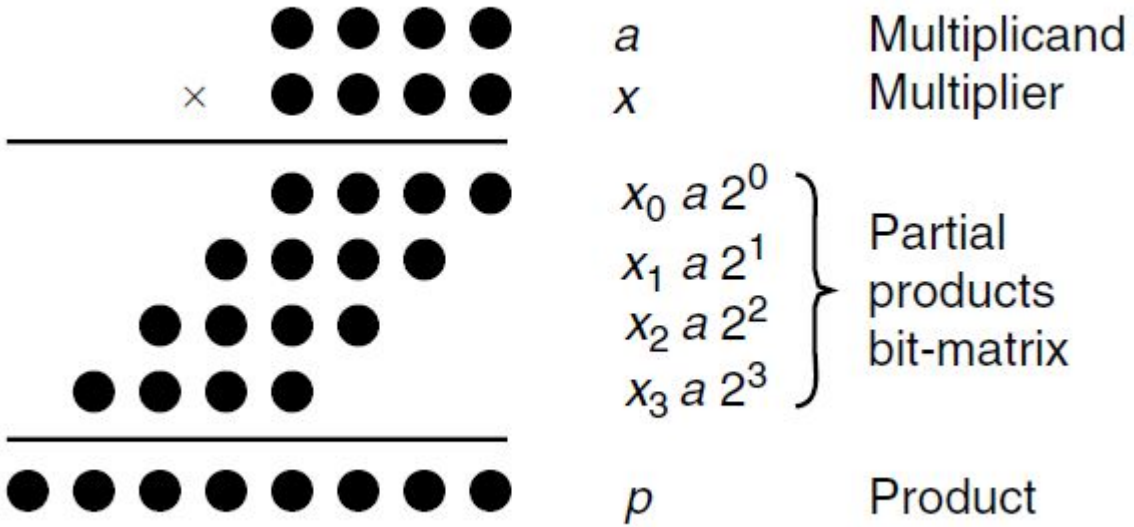
Multiplication, Multipliers and Booth's algorithm

Multiplication: SHIFT/ADD ALGORITHMS

a	Multiplicand	$a_{k-1}a_{k-2}\cdots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2}\cdots x_1x_0$
p	Product $(a \times x)$	$p_{2k-1}p_{2k-2}\cdots p_1p_0$



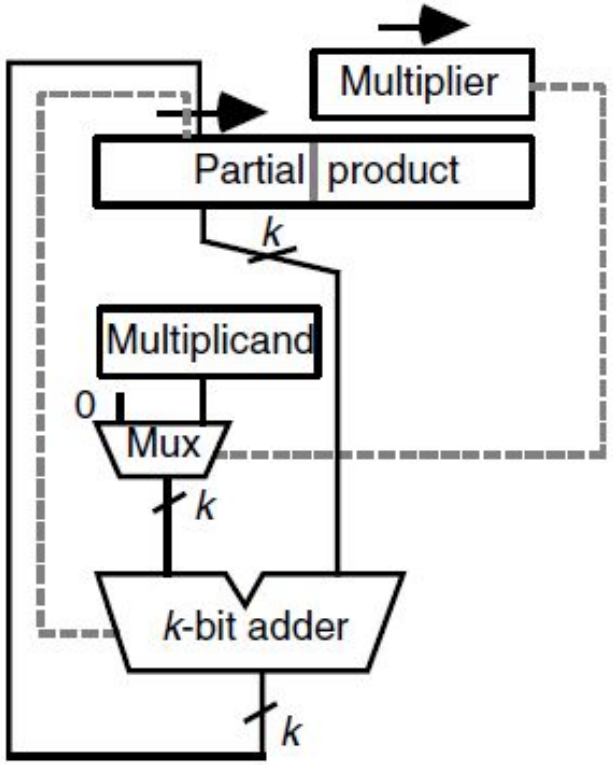
Since x_j is in {0, 1}, each term x_ja is either 0 or a. Thus, the problem of binary multiplication reduces to adding a set of numbers, each of which is 0 or a shifted version of the multiplicand a.

(a) Right-shift algorithm

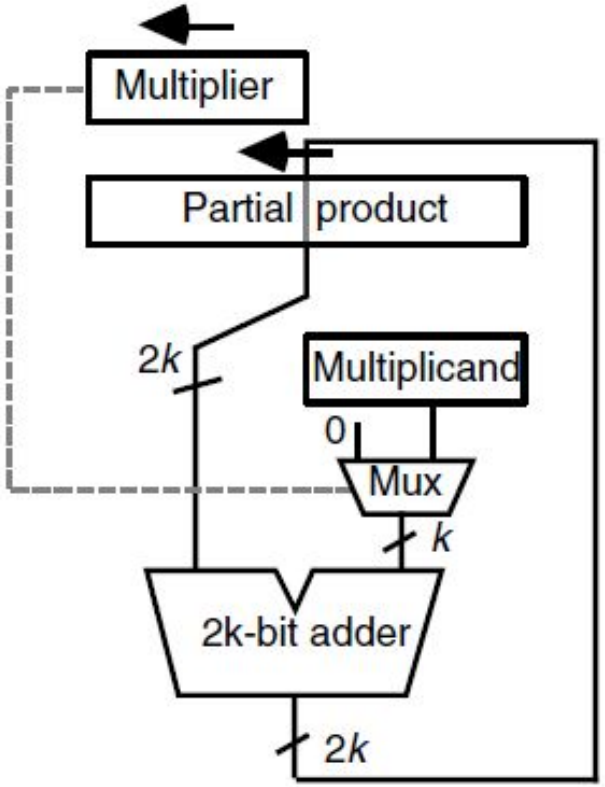
a			1	0	1	0			
x			1	0	1	1			
$p^{(0)}$			0	0	0	0			
$+x_0a$			1	0	1	0			
$2p^{(1)}$	0	1	0	1	0				
$p^{(1)}$		0	1	0	1	0			
$+x_1a$		1	0	1	0				
$2p^{(2)}$	0	1	1	1	1	0			
$p^{(2)}$		0	1	1	1	1	0		
$+x_2a$		0	0	0	0				
$2p^{(3)}$	0	0	1	1	1	1	0		
$p^{(3)}$		0	0	1	1	1	1	0	
$+x_3a$		1	0	1	0				
$2p^{(4)}$	0	1	1	0	1	1	1	0	
$p^{(4)}$		0	1	1	0	1	1	1	0

(b) Left-shift algorithm

a					1	0	1	0	
x					1	0	1	1	
$p^{(0)}$					0	0	0	0	
$2p^{(0)}$				0	0	0	0	0	
$+x_3a$					1	0	1	0	
$p^{(1)}$				0	1	0	1	0	
$2p^{(1)}$			0	1	0	1	0	0	
$+x_2a$					0	0	0	0	
$p^{(2)}$			0	1	0	1	0	0	
$2p^{(2)}$		0	1	0	1	0	0	0	
$+x_1a$					1	0	1	0	
$p^{(3)}$		0	1	1	0	0	1	0	
$2p^{(3)}$	0	1	1	0	0	1	0	0	
$+x_0a$					1	0	1	0	
$p^{(4)}$	0	1	1	0	1	1	1	0	



(a) Right shift



(b) Left shift

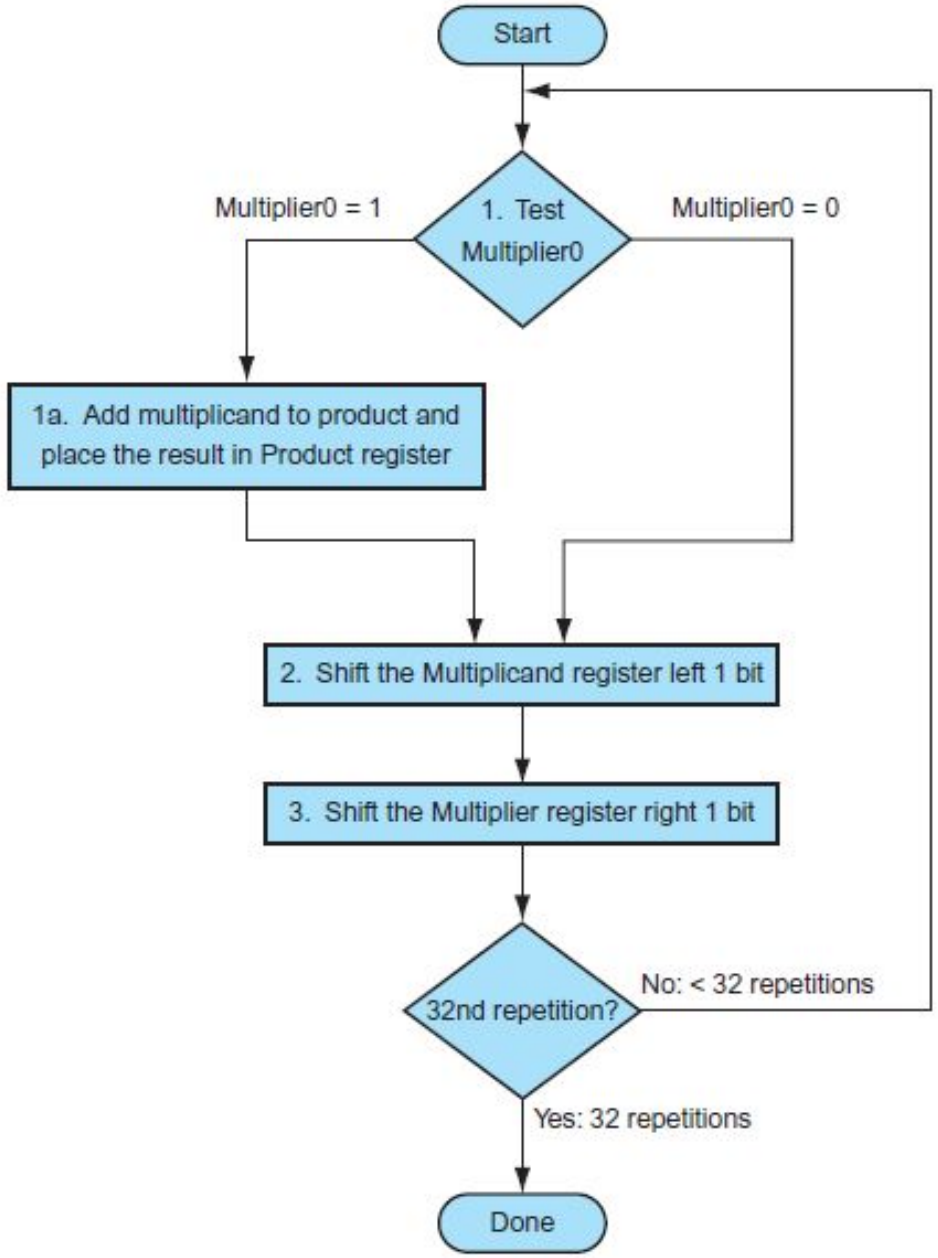
(a) Right-shift algorithm

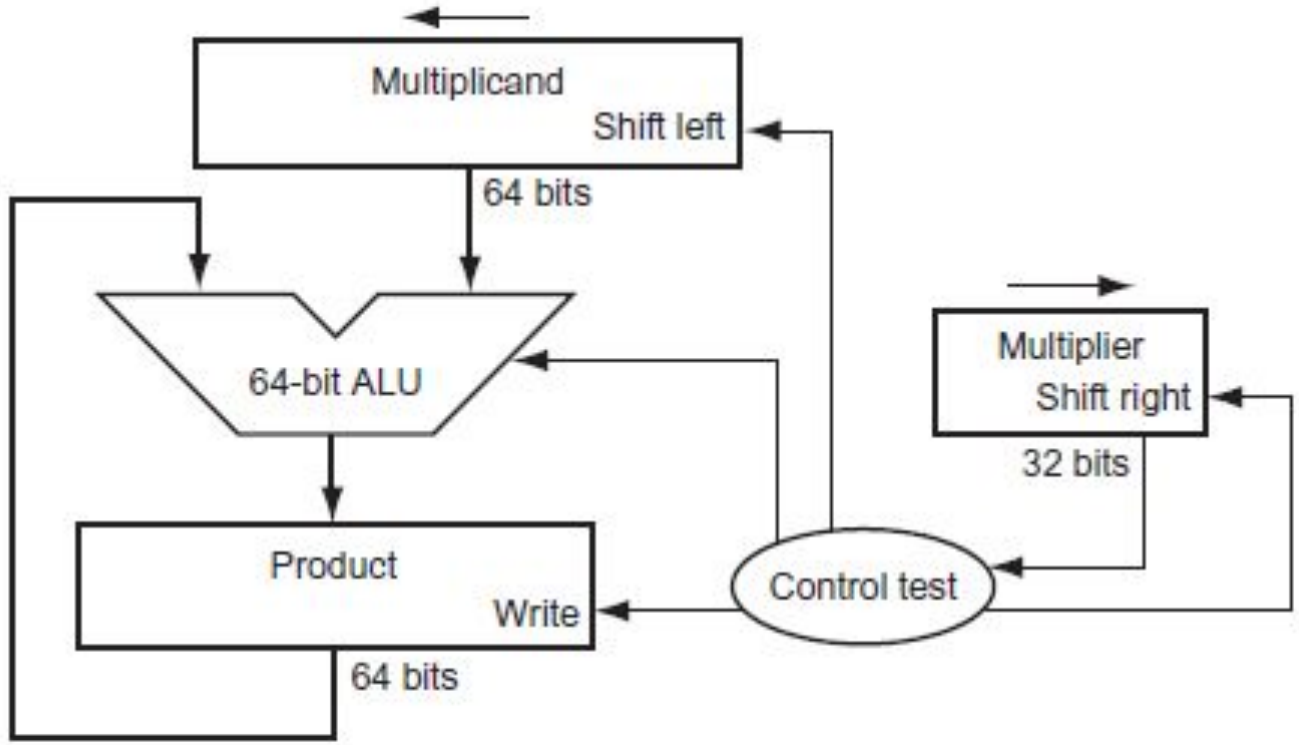
a		1	0	1	0			
x		1	0	1	1			
$p^{(0)}$		0	0	0	0			
$+x_0a$		1	0	1	0			
$2p^{(1)}$	0	1	0	1	0			
$p^{(1)}$		0	1	0	1	0		
$+x_1a$		1	0	1	0			
$2p^{(2)}$	0	1	1	1	1	0		
$p^{(2)}$		0	1	1	1	1	0	
$+x_2a$		0	0	0	0			
$2p^{(3)}$	0	0	1	1	1	1	0	
$p^{(3)}$		0	0	1	1	1	1	0
$+x_3a$		1	0	1	0			
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

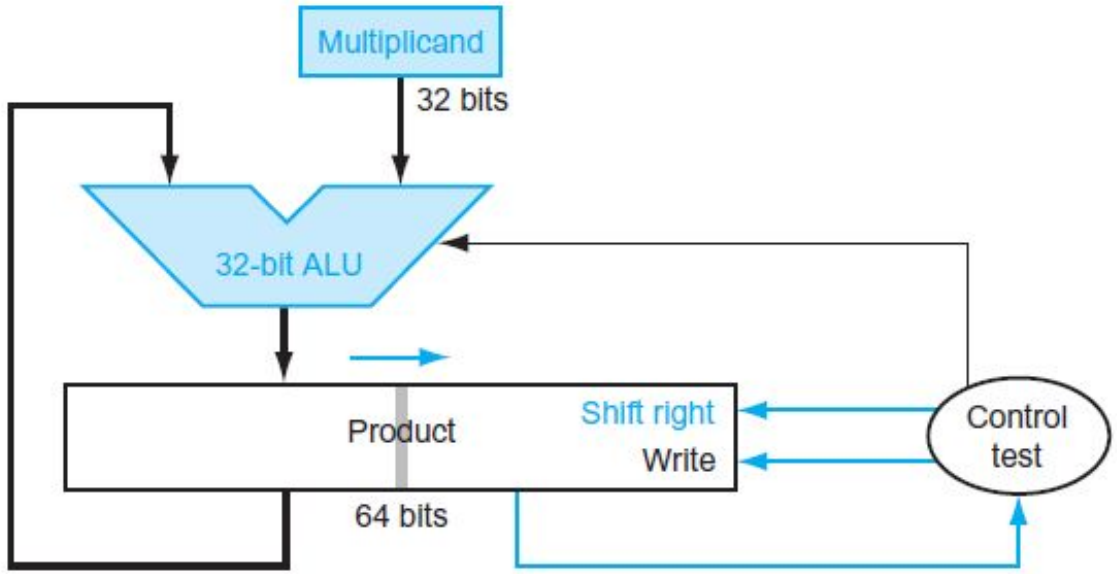
(b) Left-shift algorithm

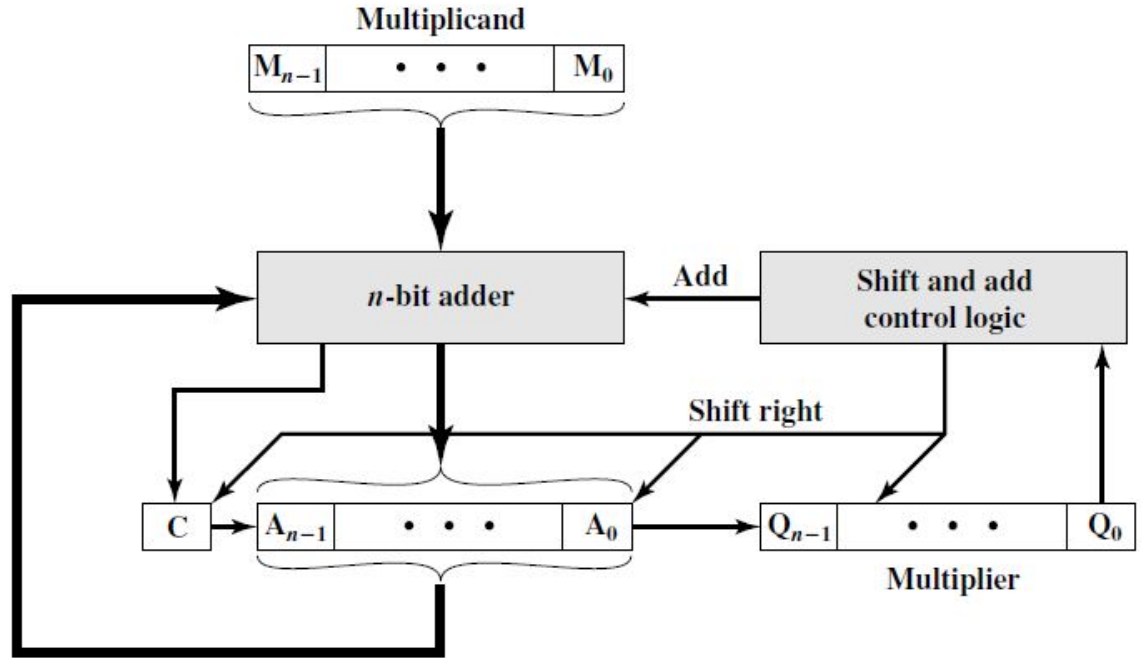
a				1	0	1	0	
x				1	0	1	1	
$p^{(0)}$				0	0	0	0	
$2p^{(0)}$			0	0	0	0	0	
$+x_3a$				1	0	1	0	
$p^{(1)}$			0	1	0	1	0	
$2p^{(1)}$		0	1	0	1	0	0	
$+x_2a$				0	0	0	0	
$p^{(2)}$		0	1	0	1	0	0	
$2p^{(2)}$	0	1	0	1	0	0	0	
$+x_1a$				1	0	1	0	
$p^{(3)}$		0	1	1	0	0	1	0
$2p^{(3)}$	0	1	1	0	0	1	0	0
$+x_0a$				1	0	1	0	
$p^{(4)}$	0	1	1	0	1	1	1	0

Multiplication with right shifts is the preferred method… Why?


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done


Multiplicand
Shift left
64 bits
64-bit ALU
Multiplier
Shift right
32 bits
Product
Write
64 bits
Control test





Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

A_3 A_2 A_1 A_0

B_3 B_2 B_1 B_0

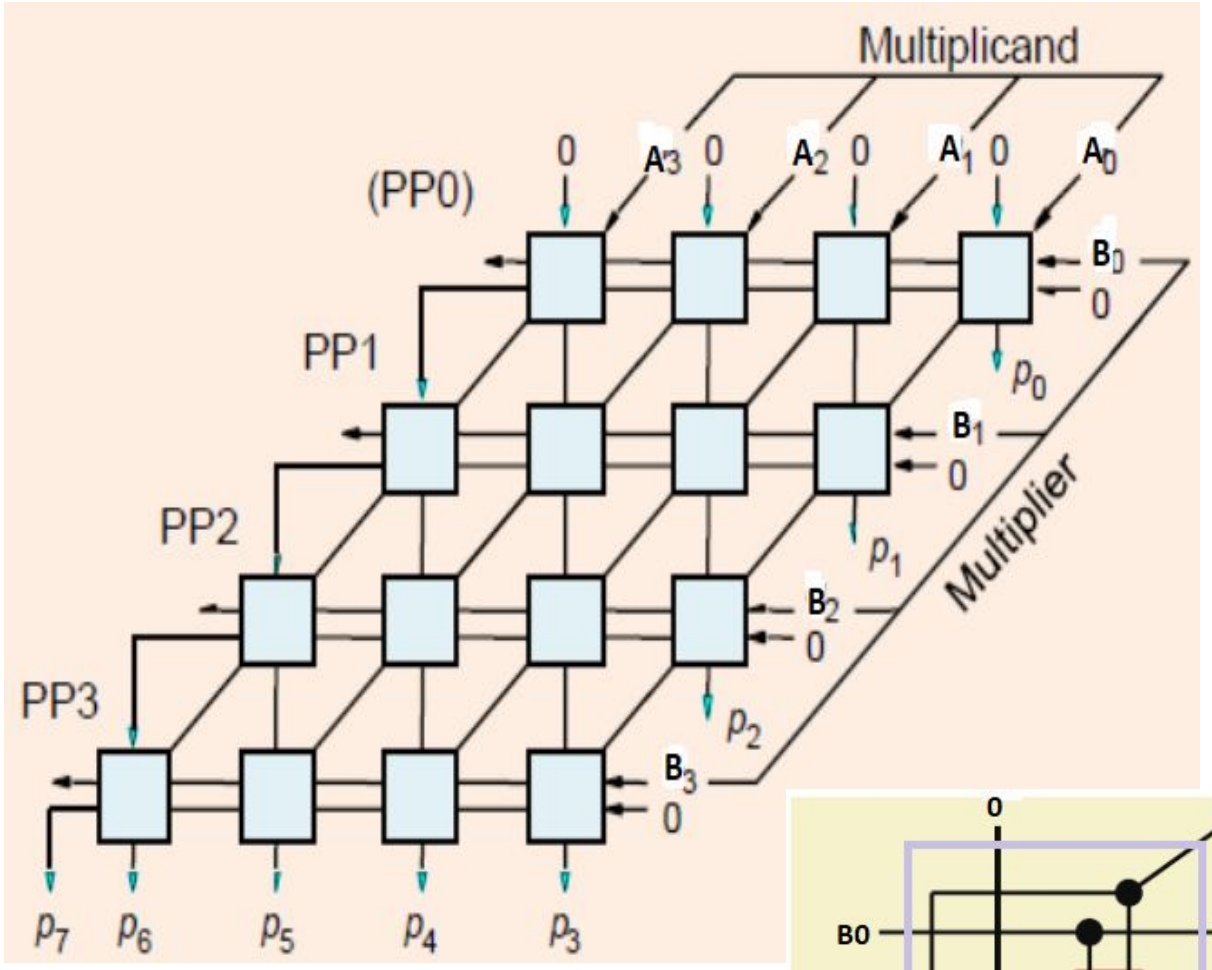
A_3B_0 A_2B_0 A_1B_0 A_0B_0

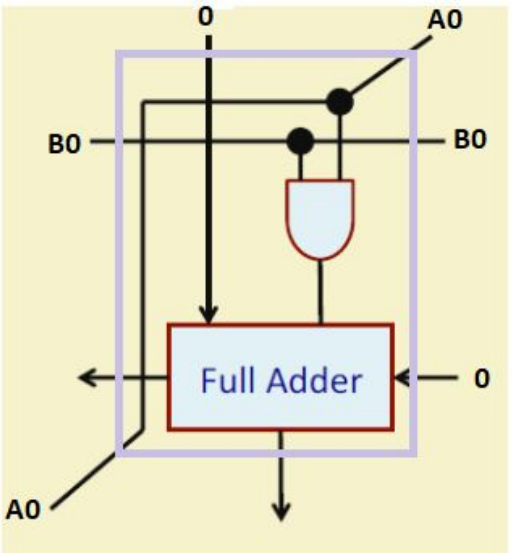
A_3B_1 A_2B_1 A_1B_1 A_0B_1

A_3B_2 A_2B_2 A_1B_2 A_0B_2

A_3B_3 A_2B_3 A_1B_3 A_0B_3

- Each A_iB_j is called a partial product.
- Generating the partial products is easy.
  - Requires just an AND gate for each partial product.
- Adding all the n-bit partial products in hardware is more difficult.





A_3 A_2 A_1 A_0

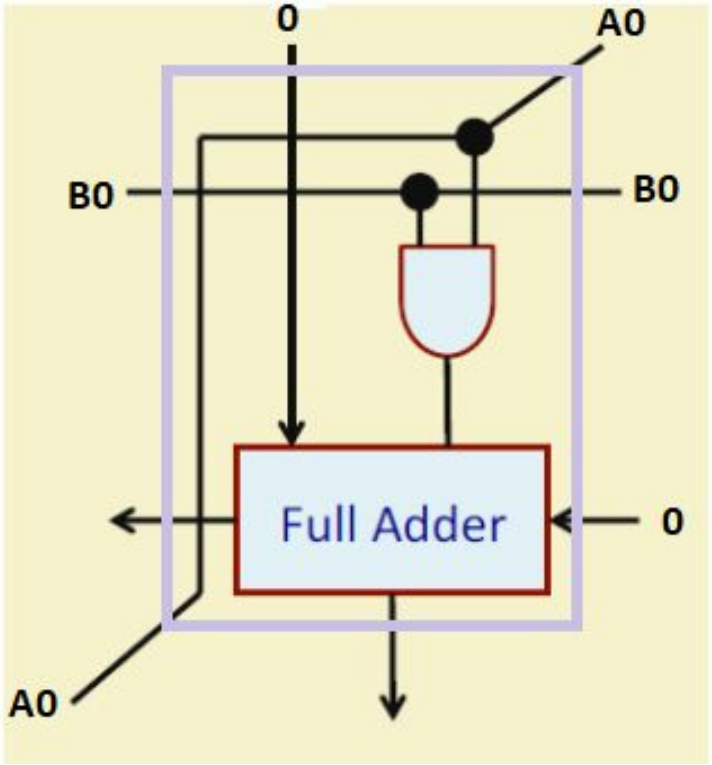
B_3 B_2 B_1 B_0

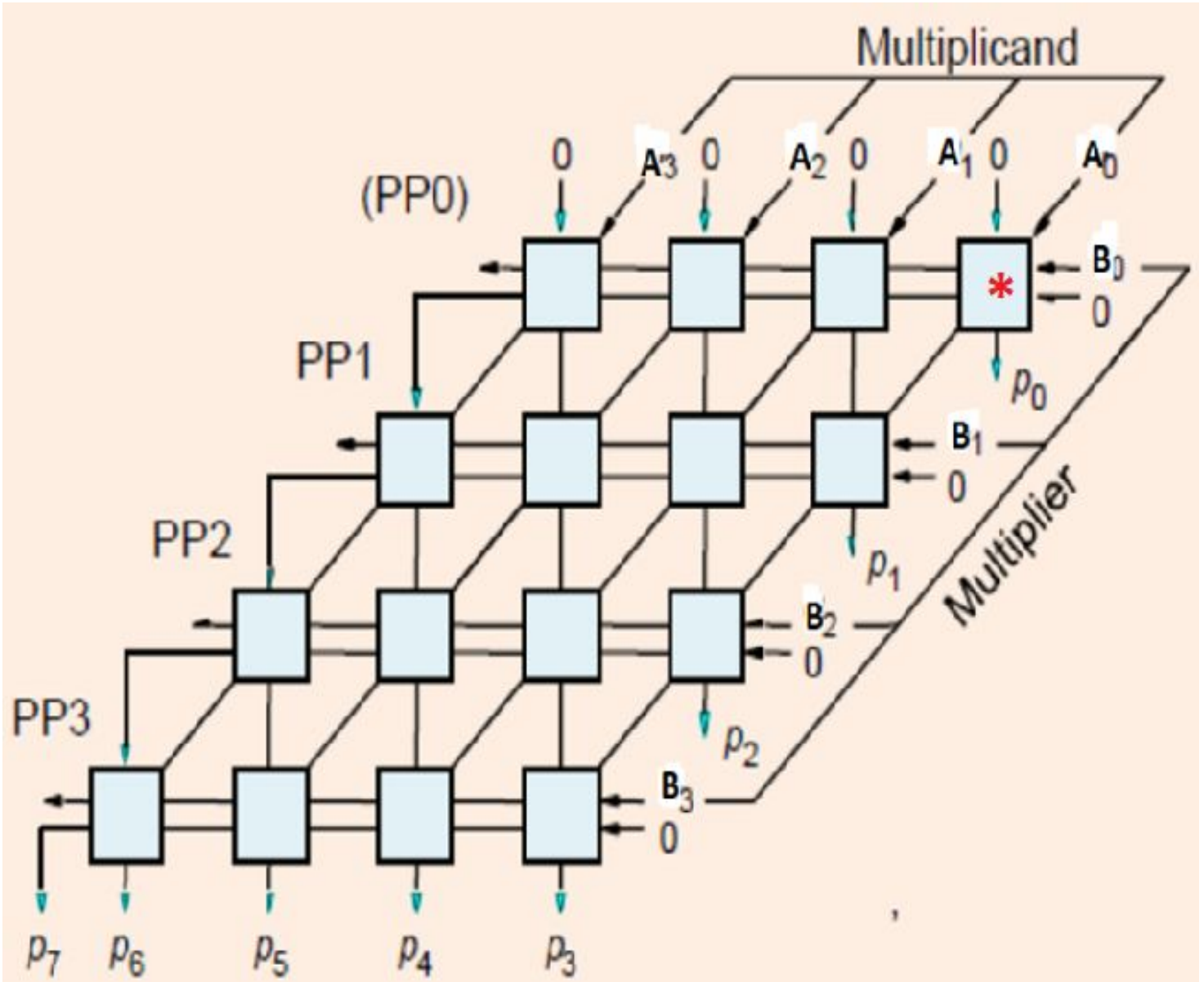
A_3B_0 A_2B_0 A_1B_0 A_0B_0

A_3B_1 A_2B_1 A_1B_1 A_0B_1

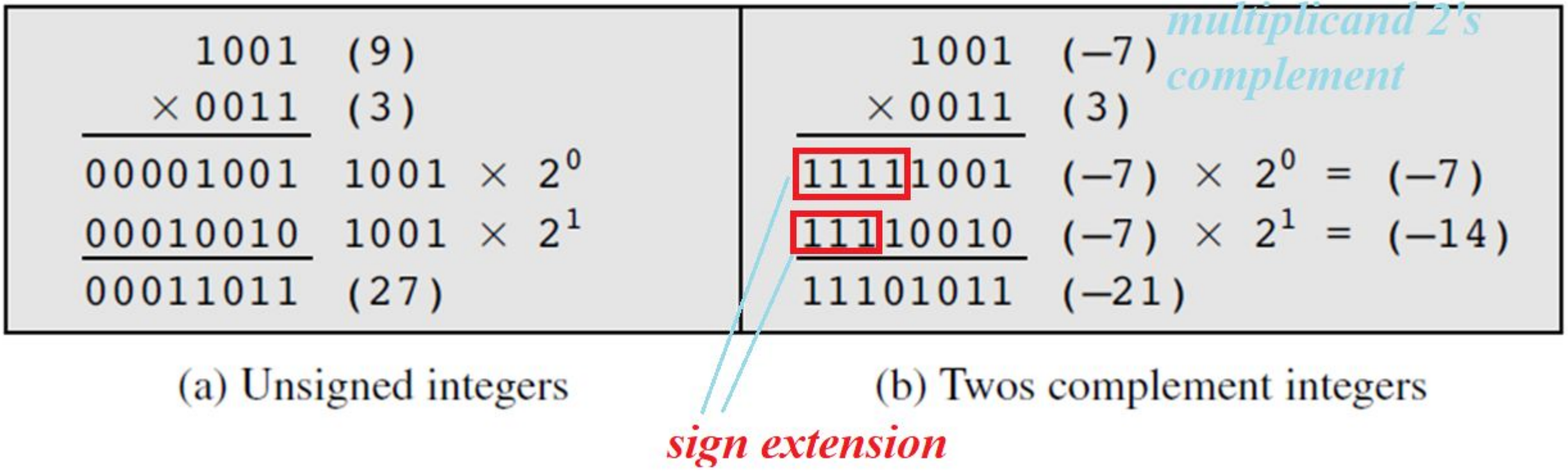
A_3B_2 A_2B_2 A_1B_2 A_0B_2

A_3B_3 A_2B_3 A_1B_3 A_0B_3





MULTIPLICATION OF SIGNED and UNSIGNED NUMBERS



Straightforward multiplication will not work

DIFFERENT…..If multiplier is Negative, will method (b) work?

Sequential multiplication of 2's-complement numbers with right-shifts (positive multiplier)

a		1	0	1	1	0			
x		0	1	0	1	1			
$p^{(0)}$		0	0	0	0	0			
$+x_0a$		1	0	1	1	0			
$2p^{(1)}$	1	1	0	1	1	0			
$p^{(1)}$		1	1	0	1	1	0		
$+x_1a$		1	0	1	1	0			
$2p^{(2)}$	1	1	0	0	0	1	0		
$p^{(2)}$		1	1	0	0	0	1	0	
$+x_2a$		0	0	0	0	0			
$2p^{(3)}$	1	1	1	0	0	0	1	0	
$p^{(3)}$		1	1	1	0	0	0	1	0
$+x_3a$		1	0	1	1	0			
$2p^{(4)}$	1	1	0	0	1	0	0	1	0
$p^{(4)}$		1	1	0	0	1	0	0	1	0
$+x_4a$		0	0	0	0	0			
$2p^{(5)}$	1	1	1	0	0	1	0	0	1	0
$p^{(5)}$		1	1	1	0	0	1	0	0	1	0

Sequential multiplication of 2's-complement numbers with right-shifts (negative multiplier)

a		1	0	1	1	0			
x		1	0	1	0	1			
$p^{(0)}$		0	0	0	0	0			
$+x_0a$		1	0	1	1	0			
$2p^{(1)}$	1	1	0	1	1	0			
$p^{(1)}$		1	1	0	1	1	0		
$+x_1a$		0	0	0	0	0			
$2p^{(2)}$	1	1	1	0	1	1	0		
$p^{(2)}$		1	1	1	0	1	1	0	
$+x_2a$		1	0	1	1	0			
$2p^{(3)}$	1	1	0	0	1	1	1	0	
$p^{(3)}$		1	1	0	0	1	1	1	0
$+x_3a$		0	0	0	0	0			
$2p^{(4)}$	1	1	1	0	0	1	1	1	0
$p^{(4)}$		1	1	1	0	0	1	1	1	0
$+(-x_4a)$		0	1	0	1	0			
$2p^{(5)}$	0	0	0	1	1	0	1	1	1	0
$p^{(5)}$		0	0	0	1	1	0	1	1	1	0

Sequential multiplication of 2's-complement numbers with right-shifts (positive multiplier)

a			1	0	1	1	0			
x			0	1	0	1	1			
$p^{(0)}$			0	0	0	0	0			
$+x_0 a$			1	0	1	1	0			
$2p^{(1)}$	1	1	0	1	1	0				
$p^{(1)}$		1	1	0	1	1	0			
$+x_1 a$		1	0	1	1	0				
$2p^{(2)}$	1	1	0	0	0	1	0			
$p^{(2)}$		1	1	0	0	0	1	0		
$+x_2 a$		0	0	0	0	0				
$2p^{(3)}$	1	1	1	0	0	0	1	0		
$p^{(3)}$		1	1	1	0	0	0	1	0	
$+x_3 a$		1	0	1	1	0				
$2p^{(4)}$	1	1	0	0	1	0	0	1	0	
$p^{(4)}$		1	1	0	0	1	0	0	1	0
$+x_4 a$		0	0	0	0	0				
$2p^{(5)}$	1	1	1	0	0	1	0	0	1	0
$p^{(5)}$		1	1	1	0	0	1	0	0	1	0

Sequential multiplication of 2's-complement numbers with right-shifts (negative multiplier)

a			1	0	1	1	0			
x			1	0	1	0	1			
$p^{(0)}$			0	0	0	0	0			
$+x_0 a$			1	0	1	1	0			
$2p^{(1)}$	1	1	0	1	1	0				
$p^{(1)}$		1	1	0	1	1	0			
$+x_1 a$		0	0	0	0	0				
$2p^{(2)}$	1	1	1	0	1	1	0			
$p^{(2)}$		1	1	1	0	1	1	0		
$+x_2 a$		1	0	1	1	0				
$2p^{(3)}$	1	1	0	0	1	1	1	0		
$p^{(3)}$		1	1	0	0	1	1	1	0	
$+x_3 a$		0	0	0	0	0				
$2p^{(4)}$	1	1	1	0	0	1	1	1	0	
$p^{(4)}$		1	1	1	0	0	1	1	1	0
$+(-x_4 a)$		0	1	0	1	0				
$2p^{(5)}$	0	0	0	1	1	0	1	1	1	0
$p^{(5)}$		0	0	0	1	1	0	1	1	1	0

Add 2's complement

If the multiplier is negative, straightforward multiplication also will not work. The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place. For example, the 4-bit decimal number -3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$$1101 \longleftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing.

- Convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed.
- Implementers have preferred to use techniques that do not require this final transformation step.

Multiplication of negative numbers

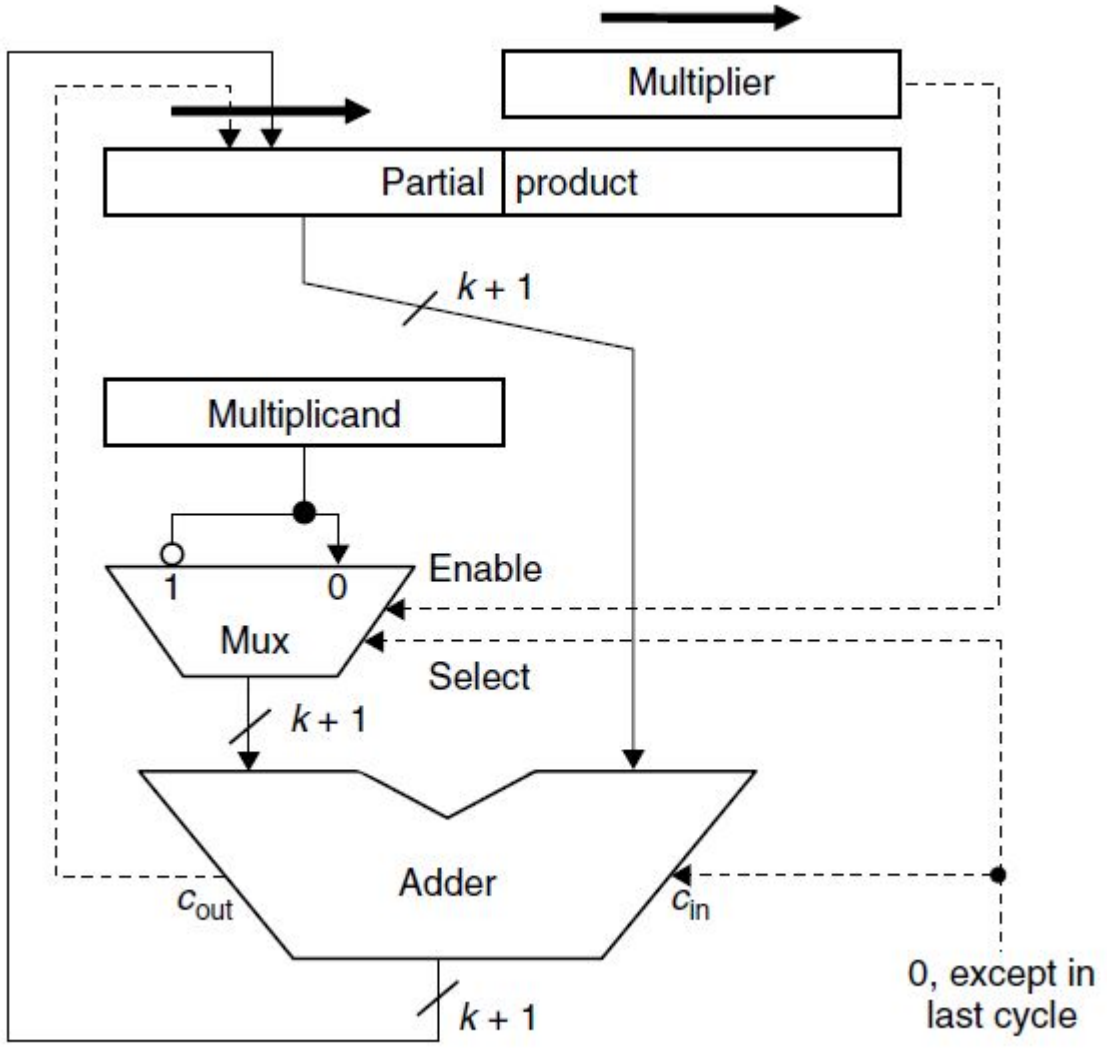
- Convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed.

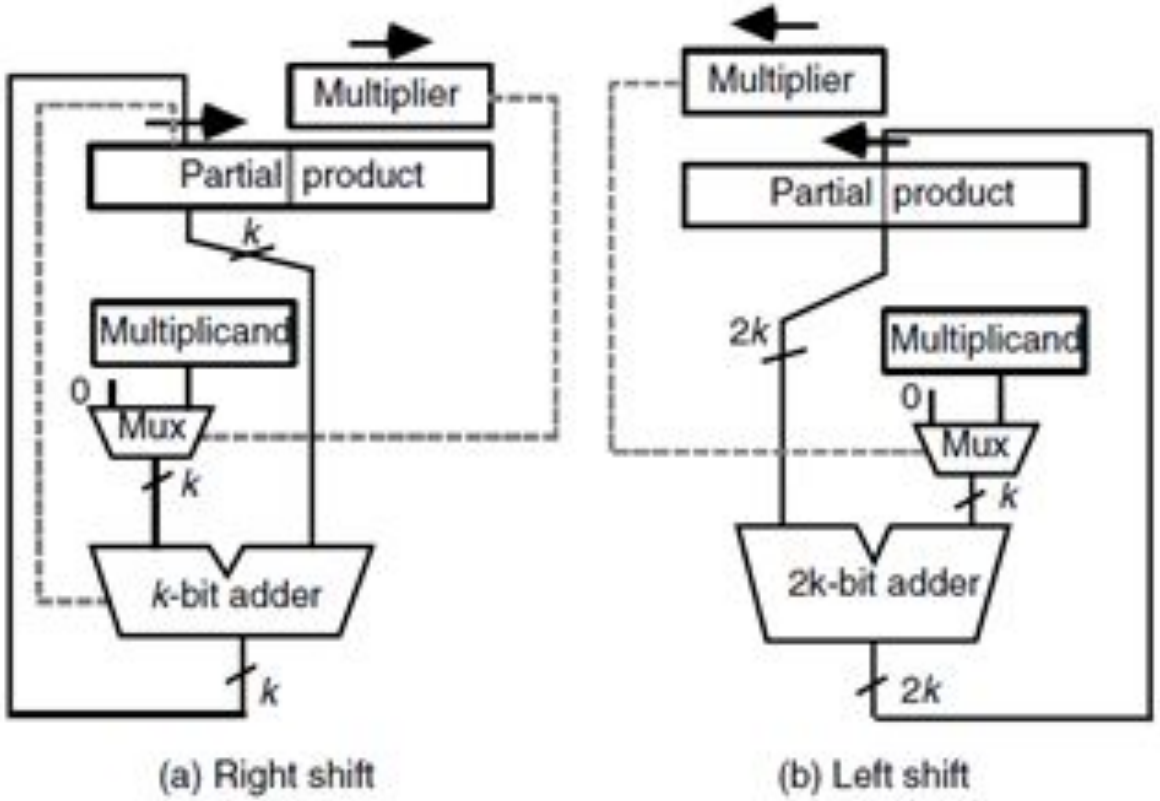
- Implementers have preferred to use techniques that do not require this final transformation step.

2’s-complement sequential hardware multiplier

Multiplicand must be sign-extended by *k* bits. We thus have a more complex adder as well as slower additions. With right shifts, on the other hand, sign extension occurs incrementally; thus the adder needs to be only 1 bit wider.

Alternatively, a *k*-bit adder can be augmented with special logic to handle the extra bit at the left.





(a) Right shift (b) Left shift

(a) Right-shift algorithm

a			1	0	1	0		
x			1	0	1	1		
$p^{(0)}$			0	0	0	0		
$+x_0a$			1	0	1	0		
$2p^{(1)}$	0	1	0	1	0			
$p^{(1)}$			0	1	0	1	0	
$+x_1a$			1	0	1	0		
$2p^{(2)}$	0	1	1	1	1	0		
$p^{(2)}$			0	1	1	1	1	0
$+x_2a$			0	0	0	0		
$2p^{(3)}$	0	0	1	1	1	1	0	
$p^{(3)}$			0	0	1	1	1	1	0
$+x_3a$			1	0	1	0		
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$			0	1	1	0	1	1	1	0

(b) Left-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$					0	0	0	0
$2p^{(0)}$				0	0	0	0	0
$+x_3a$					1	0	1	0
$p^{(1)}$				0	1	0	1	0
$2p^{(1)}$			0	1	0	1	0	0
$+x_2a$					0	0	0	0
$p^{(2)}$			0	1	0	1	0	0
$2p^{(2)}$		0	1	0	1	0	0	0
$+x_1a$					1	0	1	0
$p^{(3)}$		0	1	1	0	0	1	0
$2p^{(3)}$	0	1	1	0	0	1	0	0
$+x_0a$					1	0	1	0
$p^{(4)}$	0	1	1	0	1	1	1	0

- Adding zeros….. Could this be avoided?

- Shifting alone is faster than addition followed by shifting, and one may take advantage of this fact to reduce the multiplication time on the average. The resulting algorithm or its associated hardware implementation will have variable delay depending on the multiplier value: the more 1s there are in the binary representation of *x*, the slower the multiplication.

Booth's algorithm

- Booth observed that whenever there are a large number of consecutive 1s in *x*, multiplication can be speeded up by replacing the corresponding sequence of additions with a subtraction at the least-significant end and an addition in the position immediately to the left of its most-significant end. In other words

$$2^j + 2^{j-1} + \cdots + 2^{i+1} + 2^i = 2^{j+1} - 2^i$$

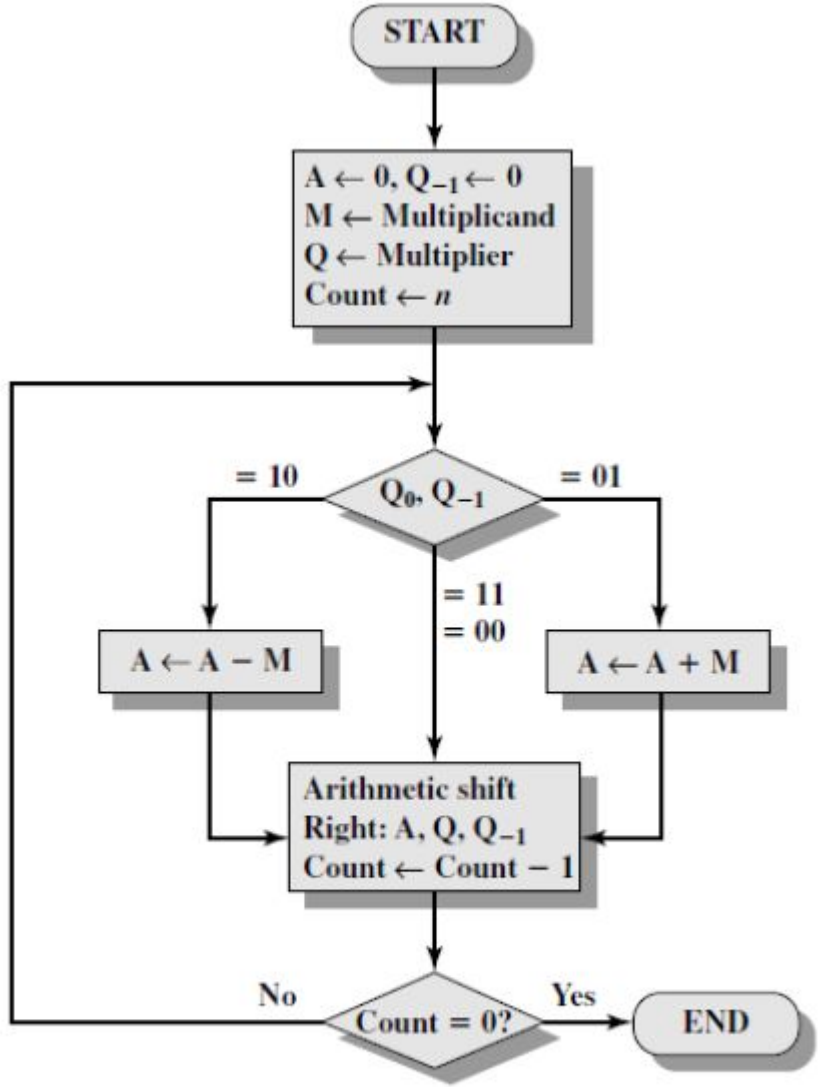
- The longer the sequence of 1s, the larger the savings achieved.
- The effect of this transformation is to change the binary number *x* with digit set [0, 1] to the binary signed digit number *y* using the digit set [−1, 1].

Radix-2 Booth's recoding.

x_i	x_{i-1}	y_i	Explanation
0	0	0	No string of 1s in sight
0	1	1	End of string of 1s in *x*
1	0	$\bar{1}$	Beginning of string of 1s in *x*
1	1	0	Continuation of string of 1s in *x*

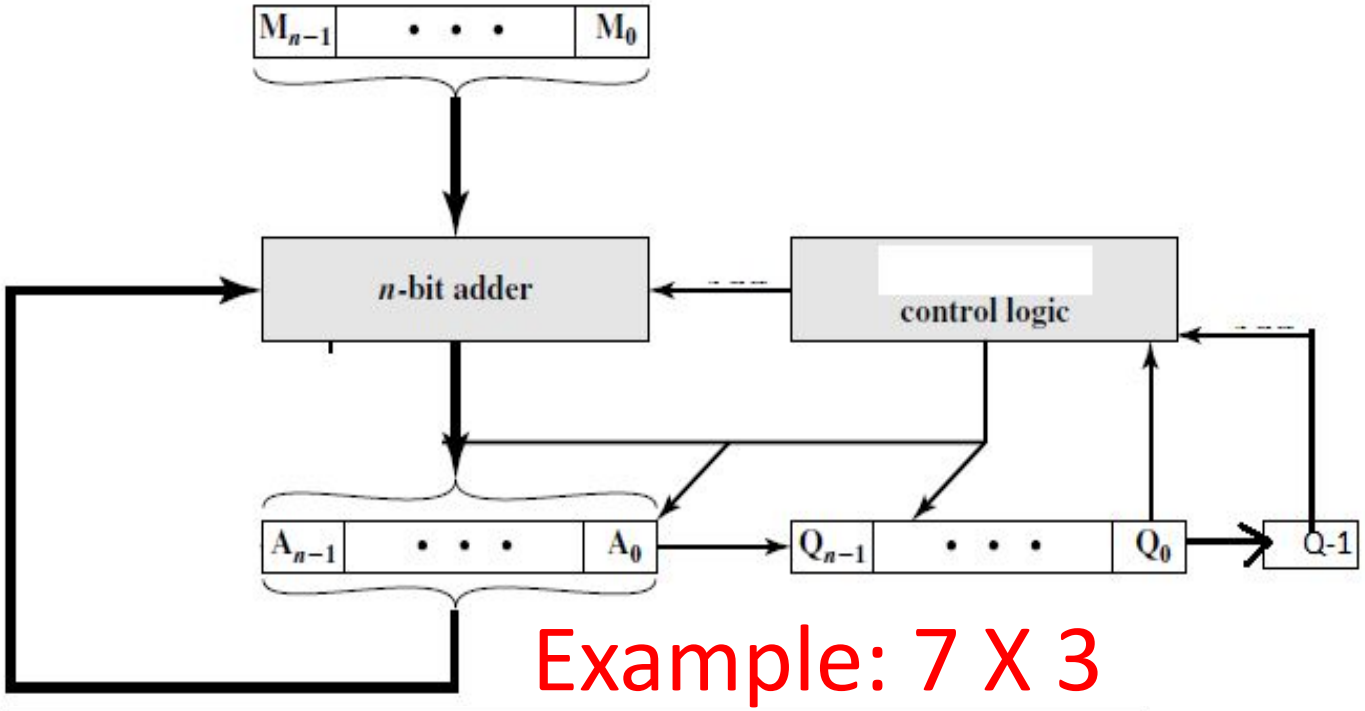
Booth's algorithm

a			1	0	1	1	0					
x			1	0	1	0	1		Multiplier			
y			$^{-}1$	1	$^{-}1$	1	$^{-}1$		Booth-recoded			
$p^{(0)}$			0	0	0	0	0					
$+y_0a$			0	1	0	1	0					
$2p^{(1)}$	0	0	1	0	1	0						
$p^{(1)}$		0	0	1	0	1	0					
$+y_1a$			1	0	1	1	0					
$2p^{(2)}$	1	1	1	0	1	1	0					
$p^{(2)}$			1	1	1	0	1	1	0			
$+y_2a$			0	1	0	1	0					
$2p^{(3)}$	0	0	0	1	1	1	1	0				
$p^{(3)}$		0	0	0	1	1	1	1	0			
$+y_3a$			1	0	1	1	0					
$2p^{(4)}$	1	1	1	0	0	1	1	1	0			
$p^{(4)}$		1	1	1	0	0	1	1	1	0		
$+y_4a$			0	1	0	1	0					
$2p^{(5)}$	0	0	0	1	1	0	1	1	1	0		
$p^{(5)}$		0	0	0	1	1	0	1	1	1	0	



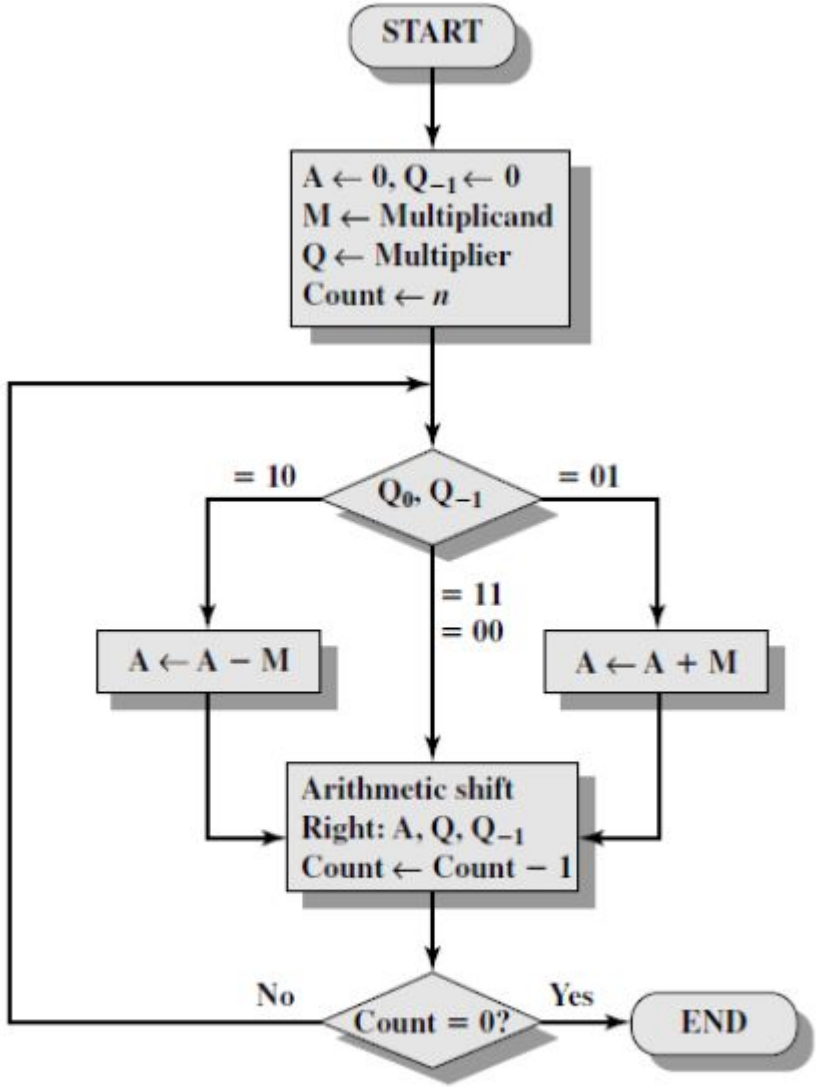
Booth's Algorithm for Twos Complement Multiplication

Booth's Algorithm

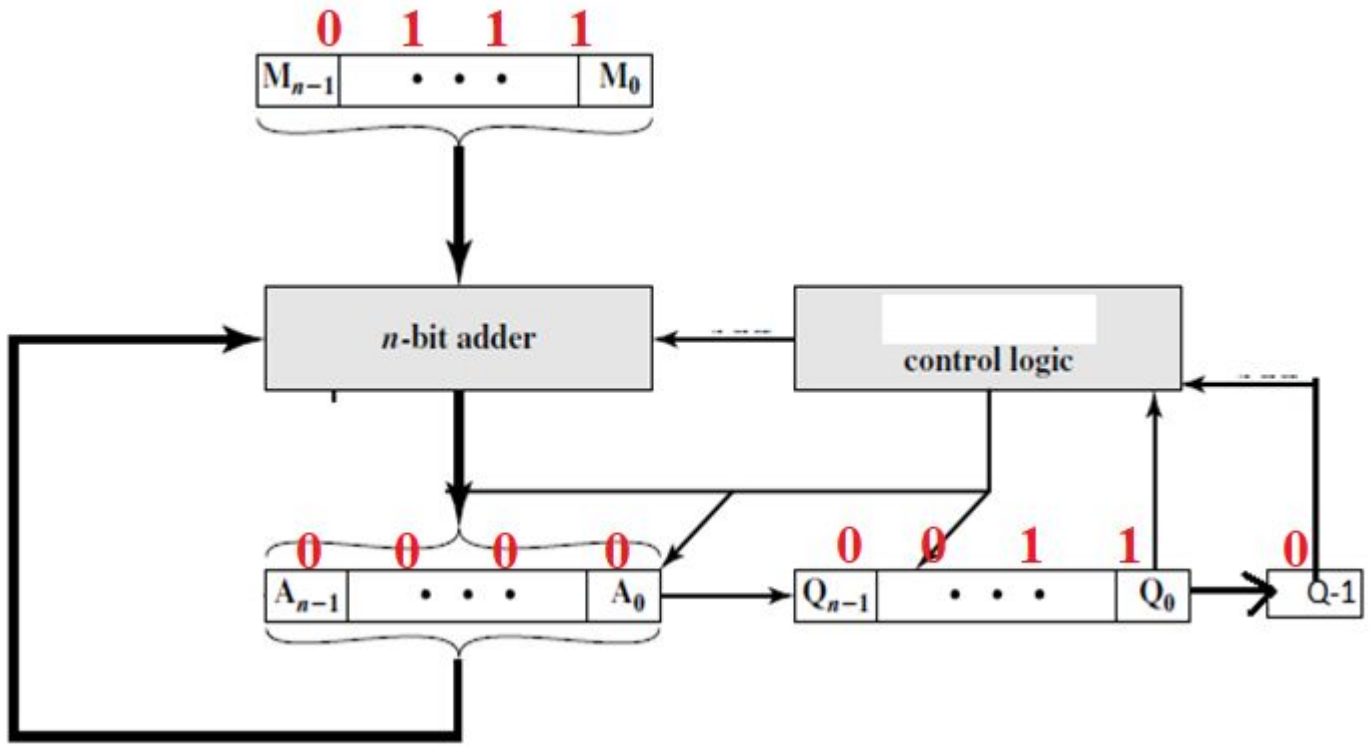


Example: 7 X 3

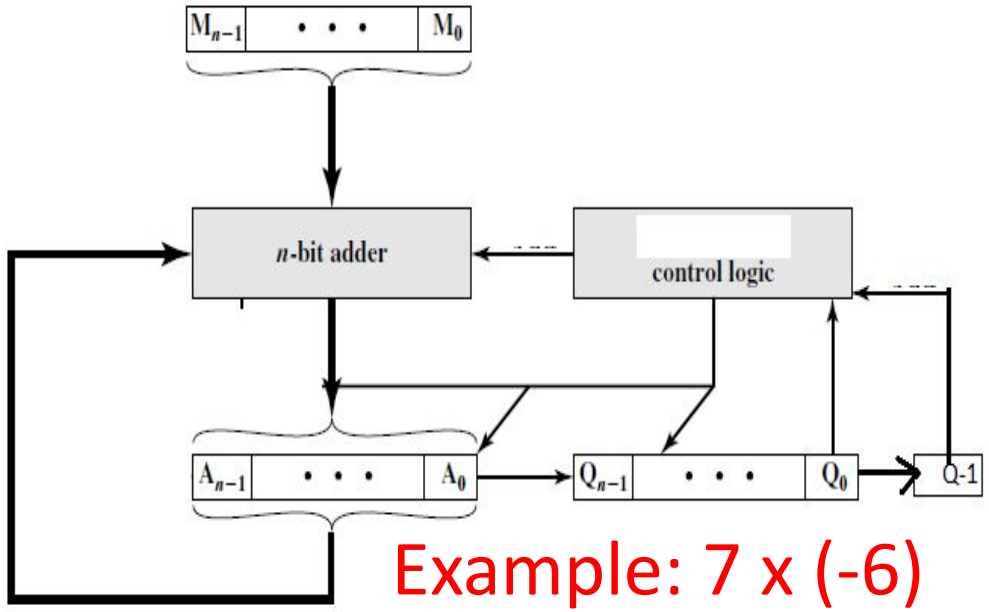
A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle



Booth's Algorithm for Twos Complement Multiplication


0 1 1 1
M_{n-1} • • • M_0
n-bit adder
control logic
0 0 0 0
A_{n-1} • • • A_0
0 0 1 1
Q_{n-1} • • • Q_0
0
Q-1

Signed Binary Multiplication



Example: 7 x (-6)

CYCLE	OPERATIONS	Content of A	Content of Q	Q_{-1}	Comments, if any	Content of M
Initial Value	Initialization	0000	1010	0	$Q_0Q_{-1}=00$	
Cycle-1						
	Arithmetic Shift right (A Q Q_{-1})	0000	0101	0	$Q_0Q_{-1}=10$	
Cycle-2	A= A-M	1001	0101	0	$Q_0Q_{-1}=10$	
	Shift right (A Q Q_{-1})	1100	1010	1	$Q_0Q_{-1}=01$	
Cycle-3	A=A+M	0011	1010	1	$Q_0Q_{-1}=01$	
	Shift right (A Q Q_{-1})	0001	1101	0	$Q_0Q_{-1}=10$	0111
Cycle-4	A=A-M	1010	1101	0	$Q_0Q_{-1}=10$	
	Shift right (A Q Q_{-1})	1101	0110	1	$Q_0Q_{-1}=01$	
Product is in: AQ pair: 11010110 in 2's complement since MSB is 1, to get magnitude, take 2's complement of AQ: 00101001 + 1 = 00101010 = 42 Answer: - 42						

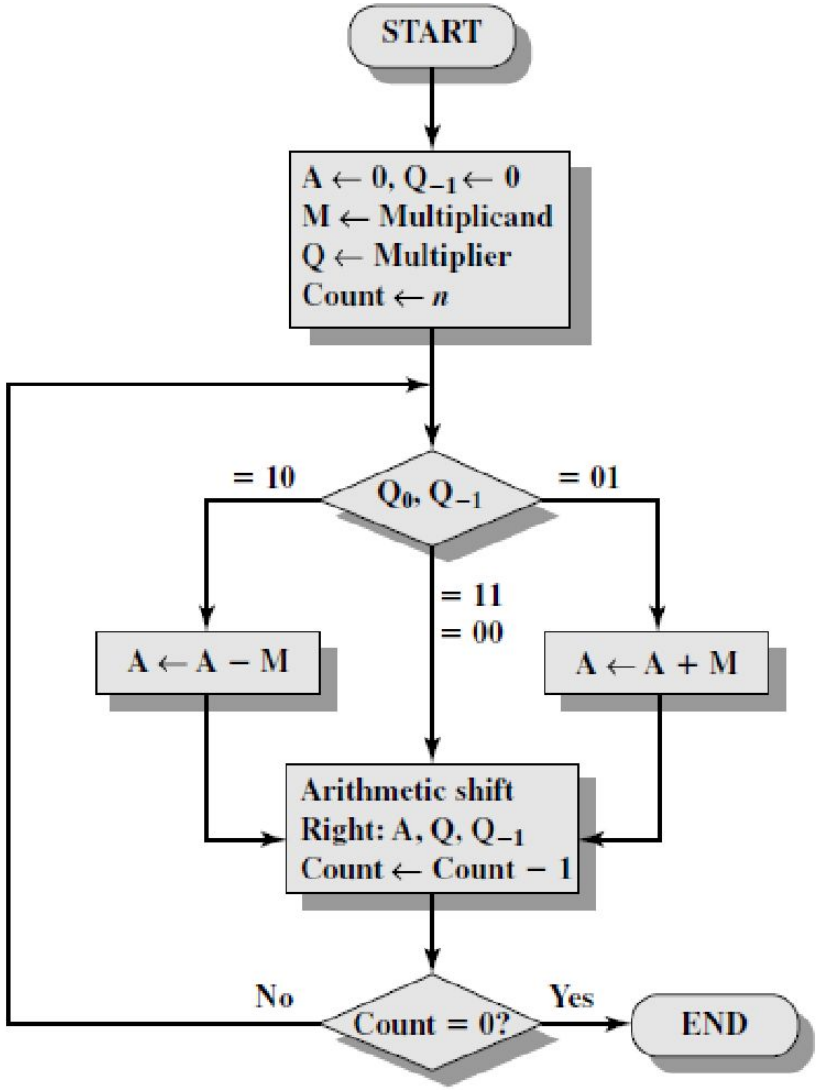
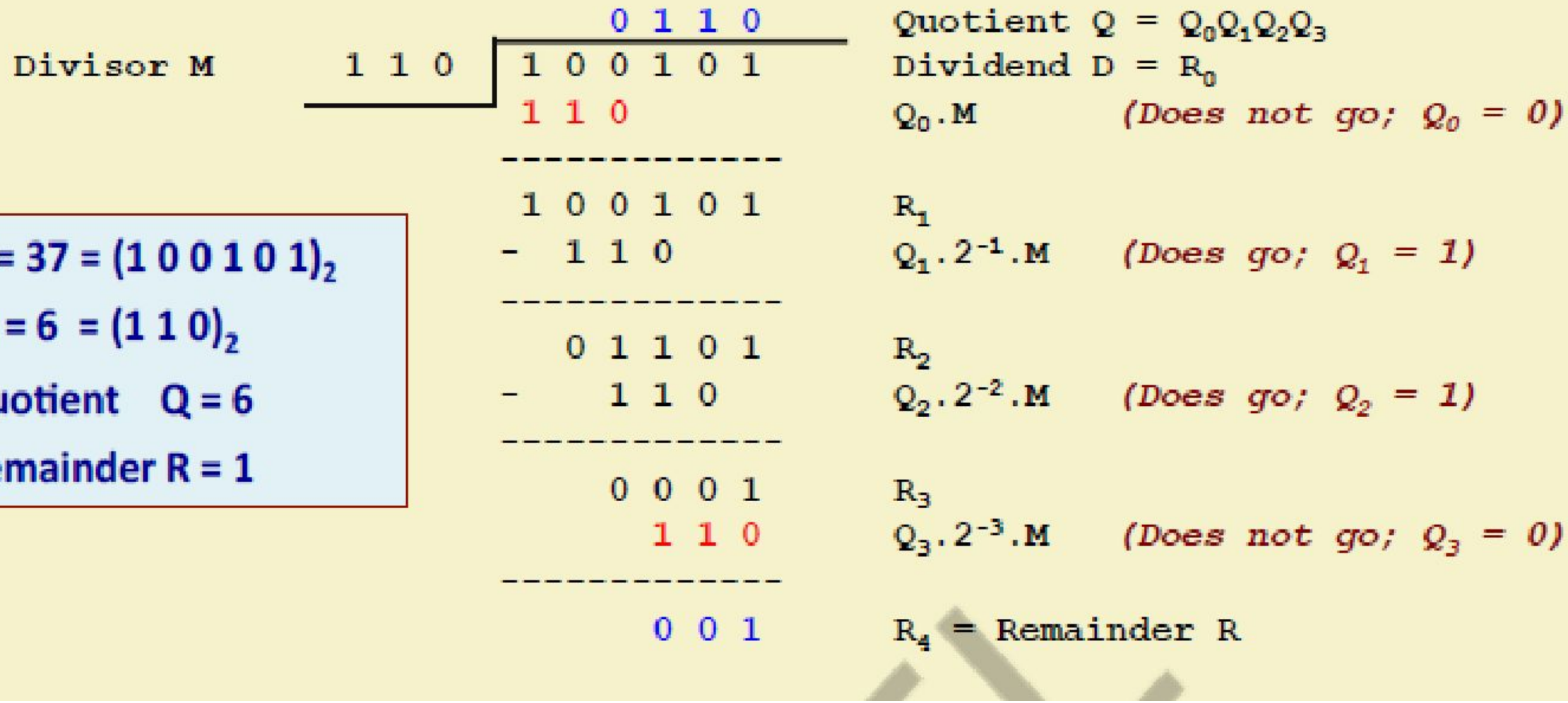


Figure 9.12 Booth's Algorithm for Twos Complement Multiplication

CYCLE	OPERATIONS	Content of A	Content of Q	Q_{-1}	Comments, if any	Content of M
Initial Value	Initialization	0000	1010	0	Q_0Q_{-1}= 00	0111
Cycle-1						
	Arithmetic Shift right ($A\ Q\ Q_{-1}$)	0000	0101	0	Q_0Q_{-1}= 10	
Cycle-2	A= A-M	1001	0101	0	Q_0Q_{-1}= 10	
	Shift right ($A\ Q\ Q_{-1}$)	1100	1010	1	Q_0Q_{-1}= 01	
Cycle-3	A=A+M	0011	1010	1	Q_0Q_{-1}= 01	
	Shift right ($A\ Q\ Q_{-1}$)	0001	1101	0	Q_0Q_{-1}= 10	
Cycle-4	A=A-M	1010	1101	0	Q_0Q_{-1}= 10	
	Shift right ($A\ Q\ Q_{-1}$)	1101	0110	1	Q_0Q_{-1}= 01	
Product is in: AQ pair: 11010110 in 2's complement since MSB is 1, to get magnitude, take 2's complement of AQ: 00101001 + 1 = 00101010 = 42 Answer: - 42						

Division

- One of the simplest division methods is the sequential digit-by-digit algorithm similar to that used in pencil-and-paper methods.



D = 37 = $(1\,0\,0\,1\,0\,1)_2$

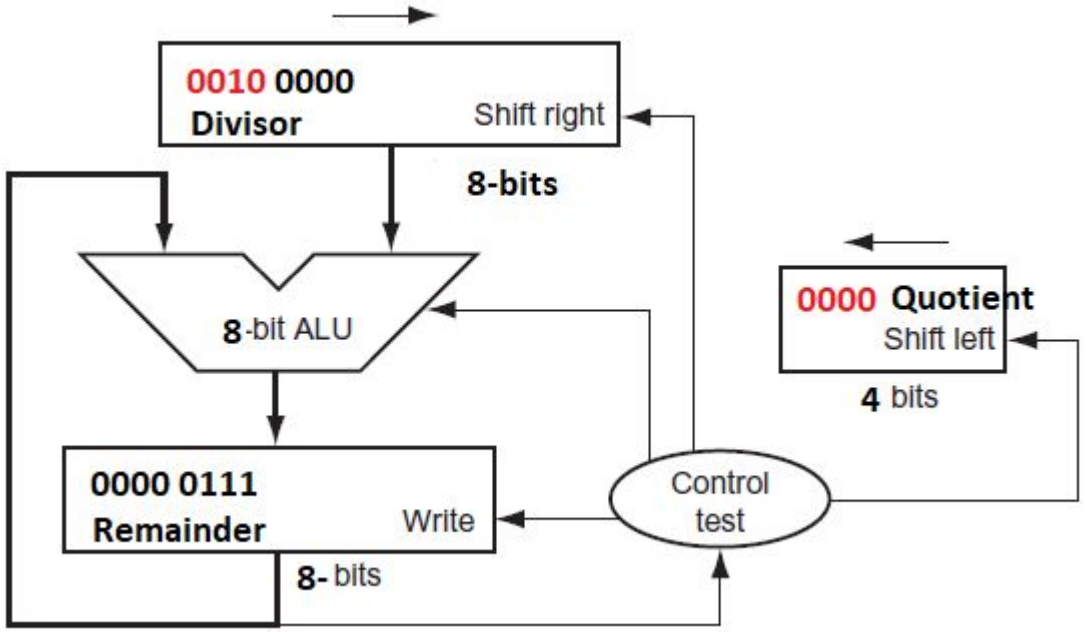
M = 6 = $(1\,1\,0)_2$

Quotient Q = 6

Remainder R = 1

Example: Divide 7 by 2

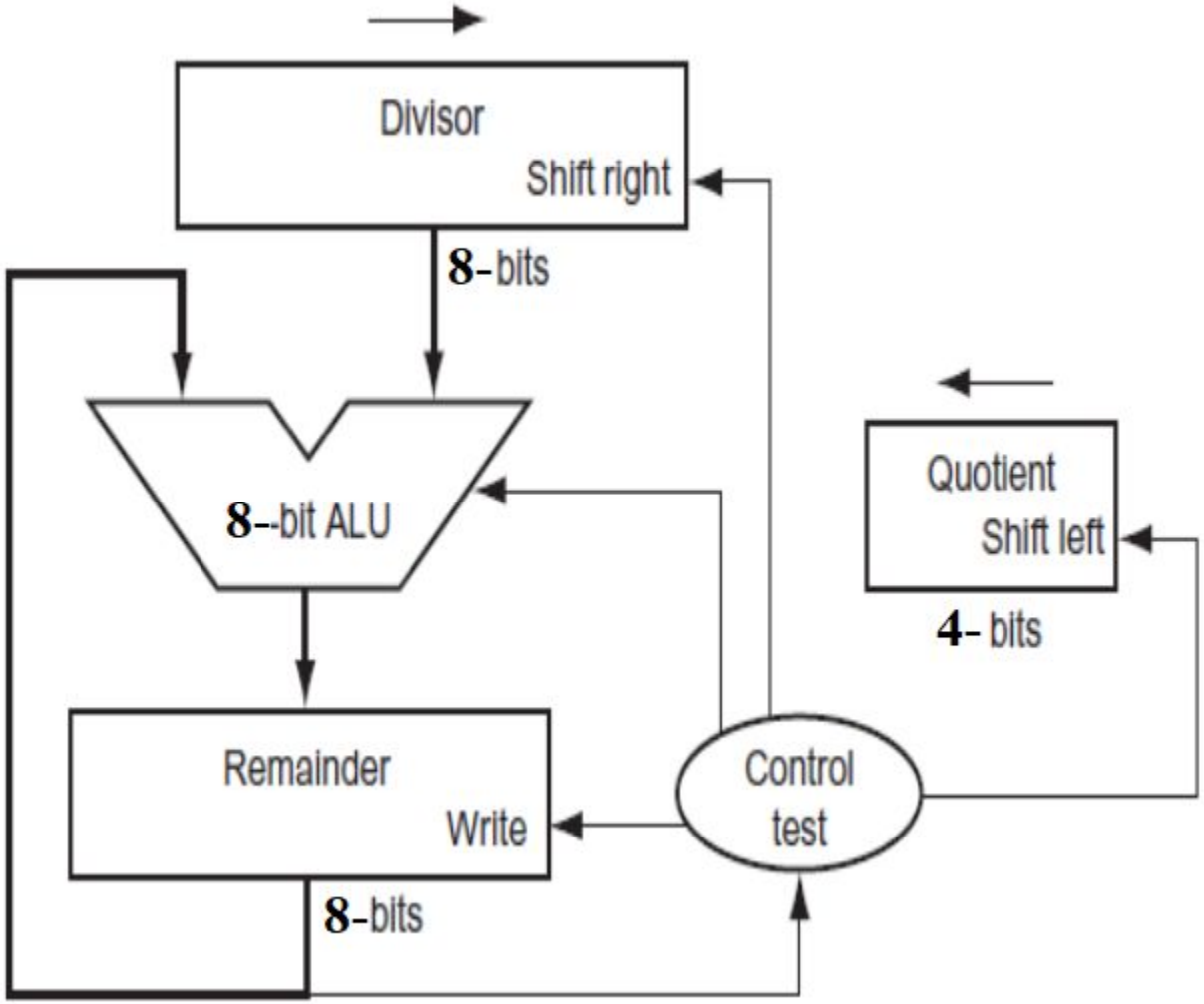
07= 0111 (Dividend) 2 = 0010 (Divisor)

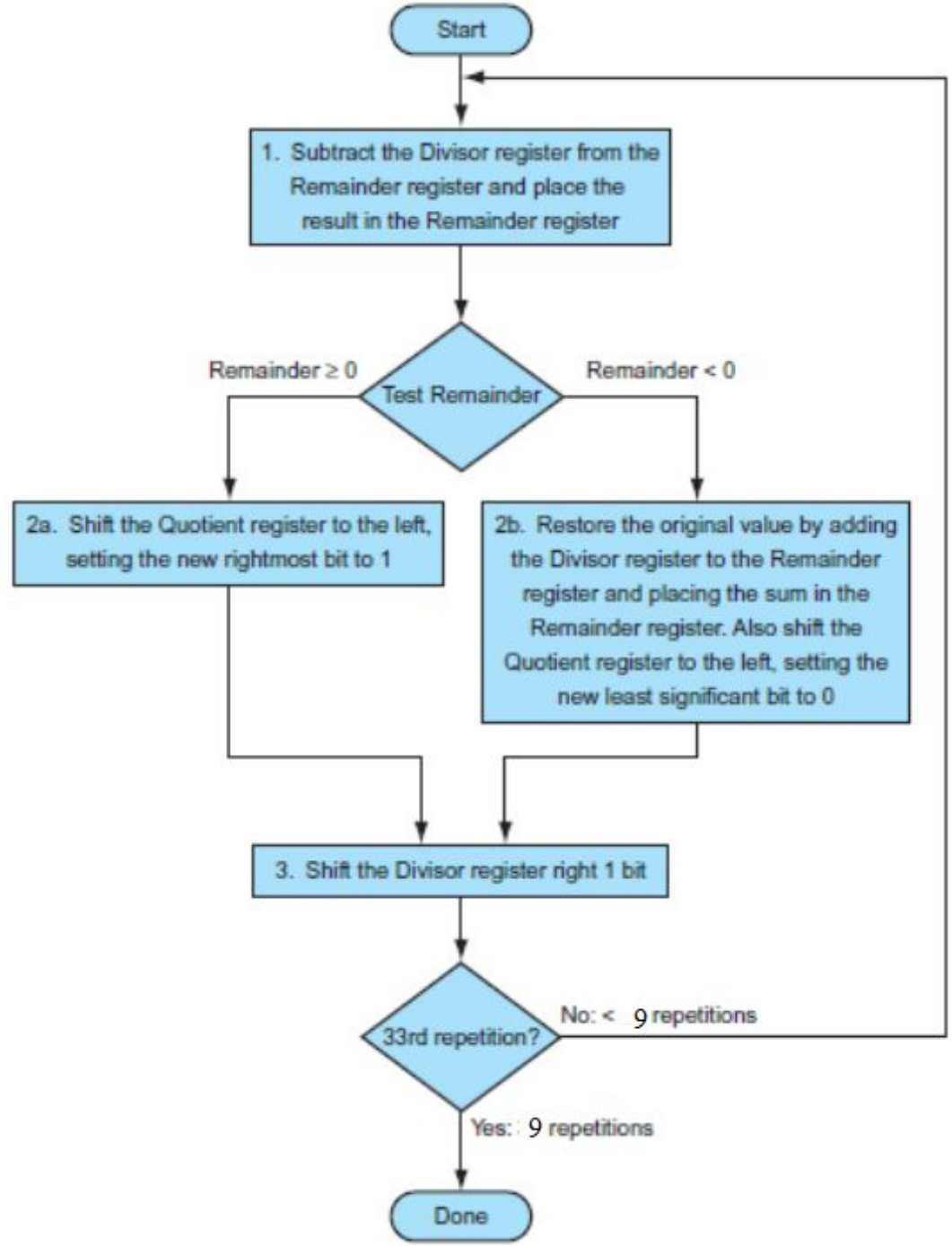


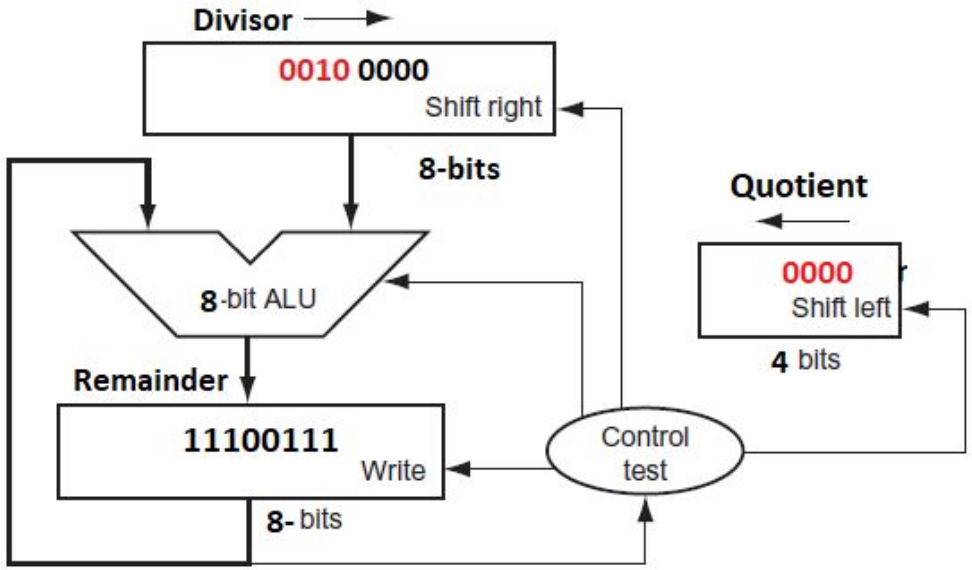
- Quotient register set to 0.
- Place the divisor in the left half of Divisor Register
- Remainder register is initialized with the dividend.
- In each iteration of the algorithm
- Move the divisor to the right one digit

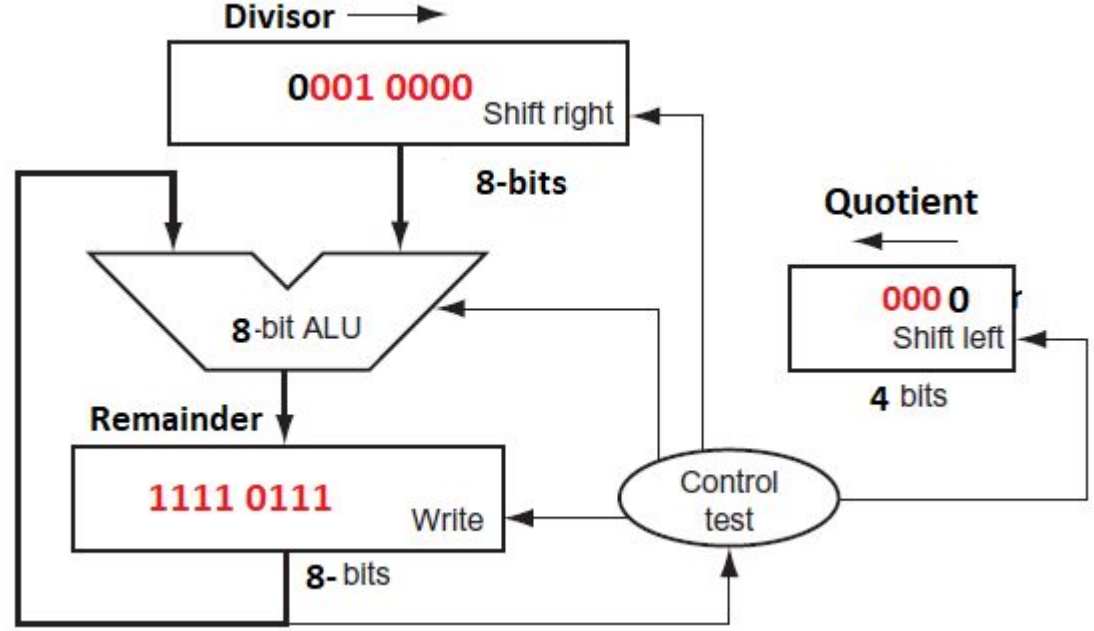
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

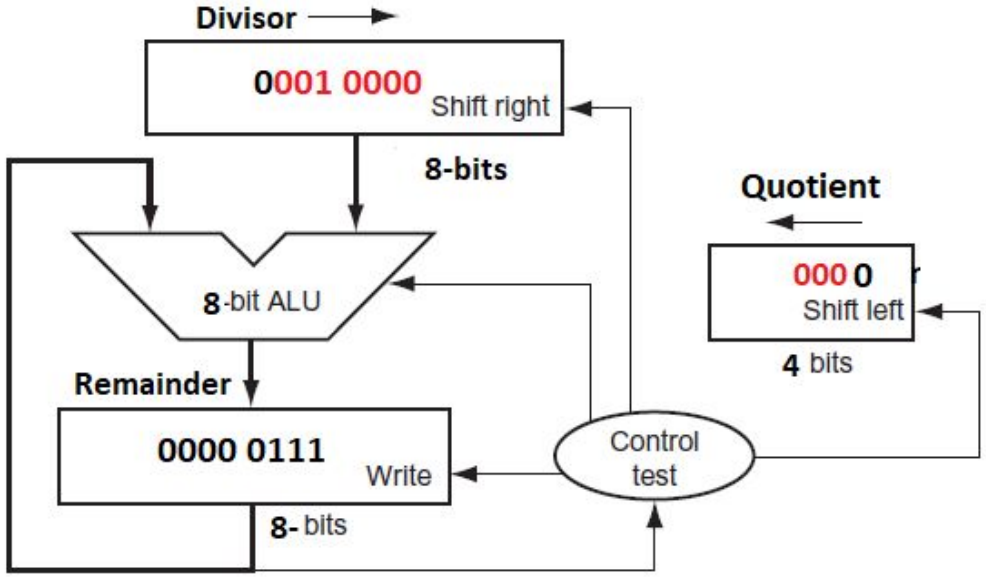
- first subtract the divisor in step 1;
- If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient
- If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient
- The divisor is shifted right and then we iterate again.
- The remainder and quotient will be found in their namesake registers after the iterations are complete.

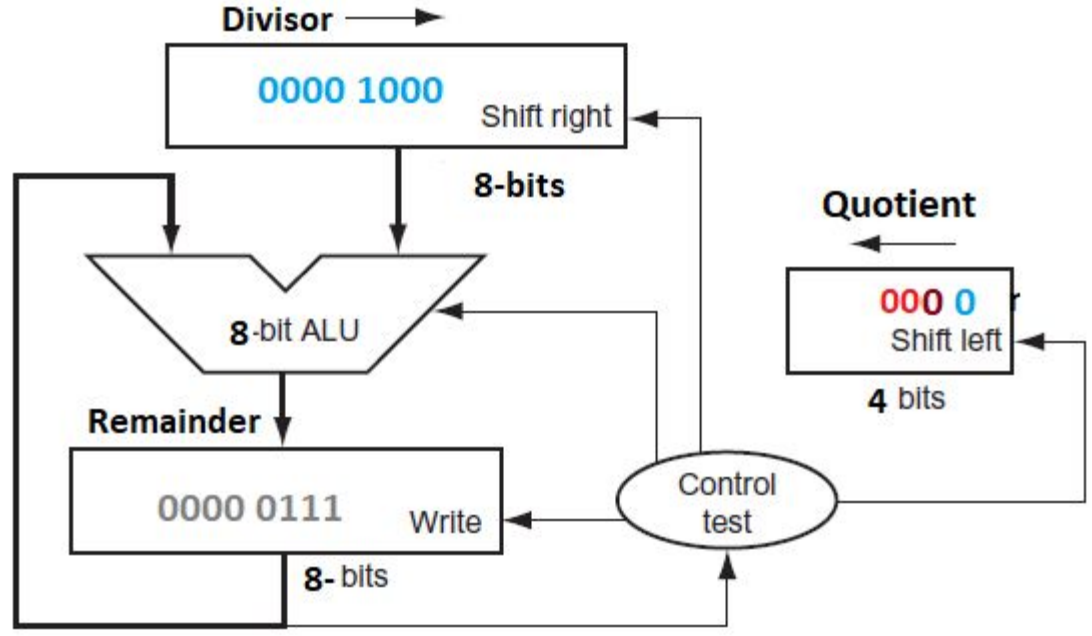

Divisor
Shift right
8-bits
8-bit ALU
Quotient
Shift left
4-bits
Remainder
Write
Control
test
8-bits


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 9 repetitions
Yes: 9 repetitions
Done


Divisor
0010 0000
Shift right
8-bits
8-bit ALU
Remainder
11100111
Write
8- bits
Control test
Quotient
0000
Shift left
4 bits

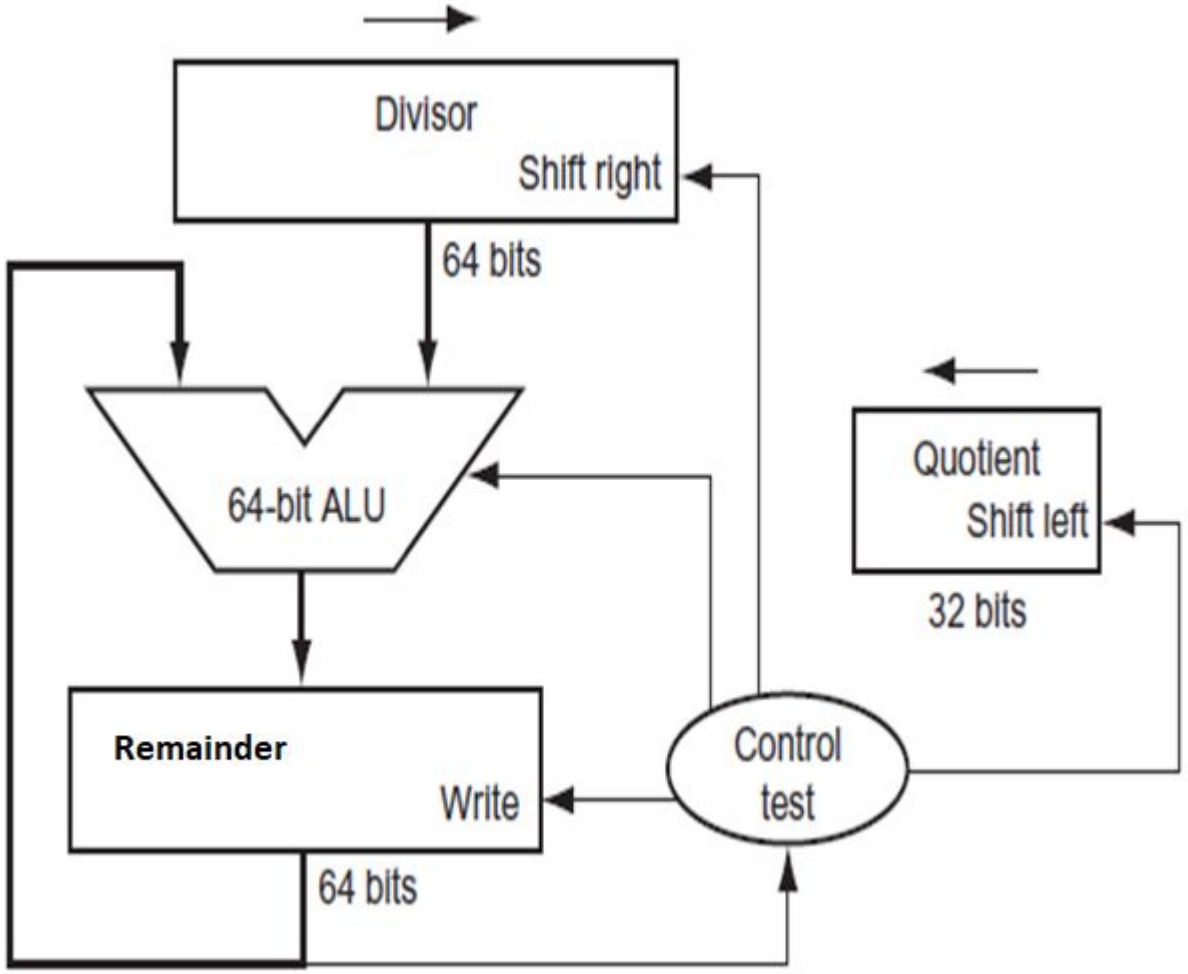

Divisor
0001 0000
Shift right
8-bits
8-bit ALU
Remainder
1111 0111
Write
8- bits
Control test
Quotient
000 0
Shift left
4 bits


Divisor
0001 0000
Shift right
8-bits
8-bit ALU
Remainder
0000 0111
Write
8- bits
Control test
Quotient
000 0
Shift left
4 bits

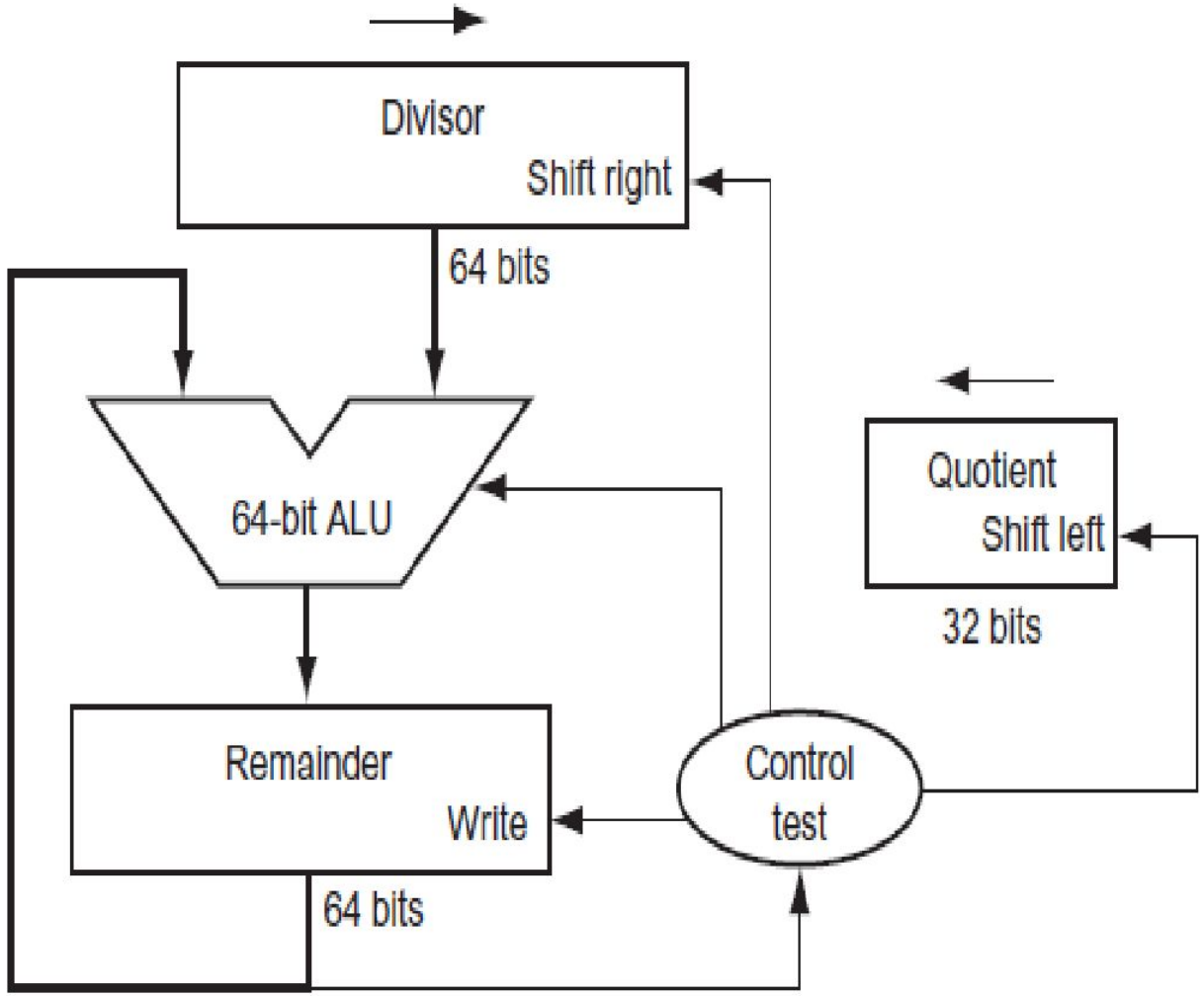

Divisor
0000 1000
Shift right
8-bits
8-bit ALU
Remainder
0000 0111
Write
8- bits
Control test
Quotient
000 0
Shift left
4 bits

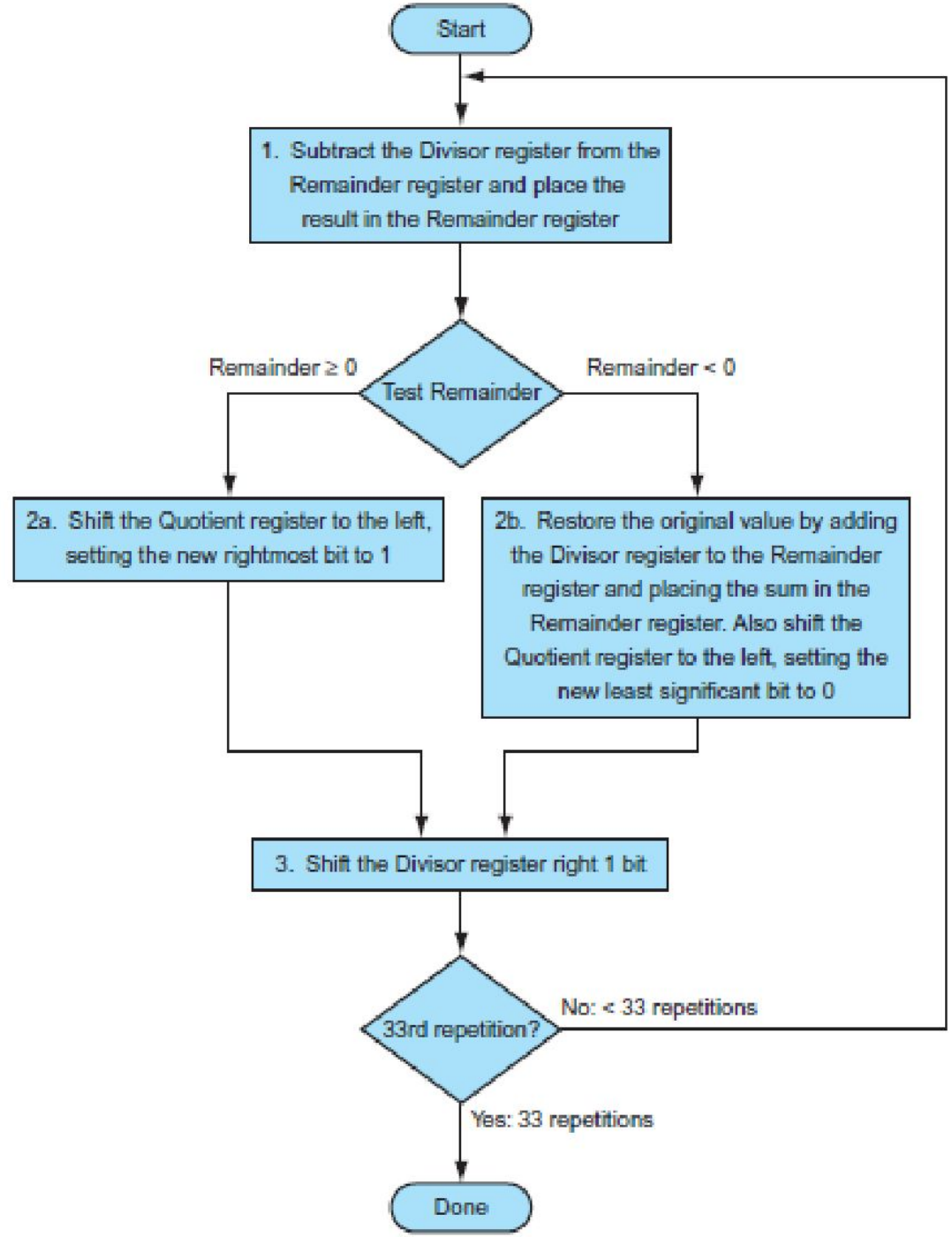
Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

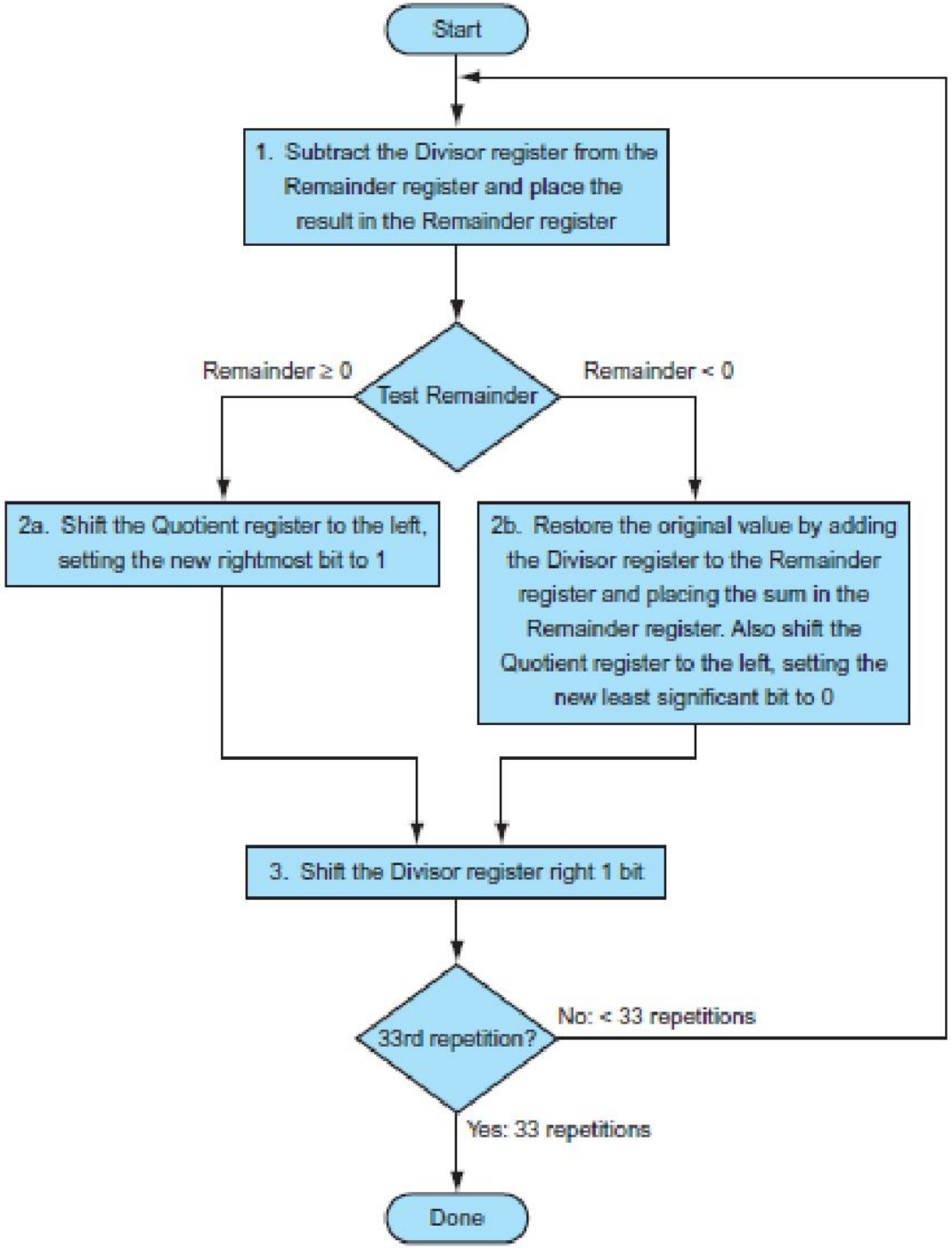
FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.



- Quotient register set to 0.
- Place the divisor in the left half of Divisor Register
- Remainder register is initialized with the dividend.
- In each iteration of the algorithm
- Move the divisor to the right one digit
- first subtract the divisor in step 1;
- If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient
- If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient
- The divisor is shifted right and then we iterate again.
- The remainder and quotient will be found in their namesake registers after the iterations are complete.


Divisor
Shift right
64 bits
64-bit ALU
Remainder
Write
64 bits
Control
test
Quotient
Shift left
32 bits


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	⓪000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	⓪000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

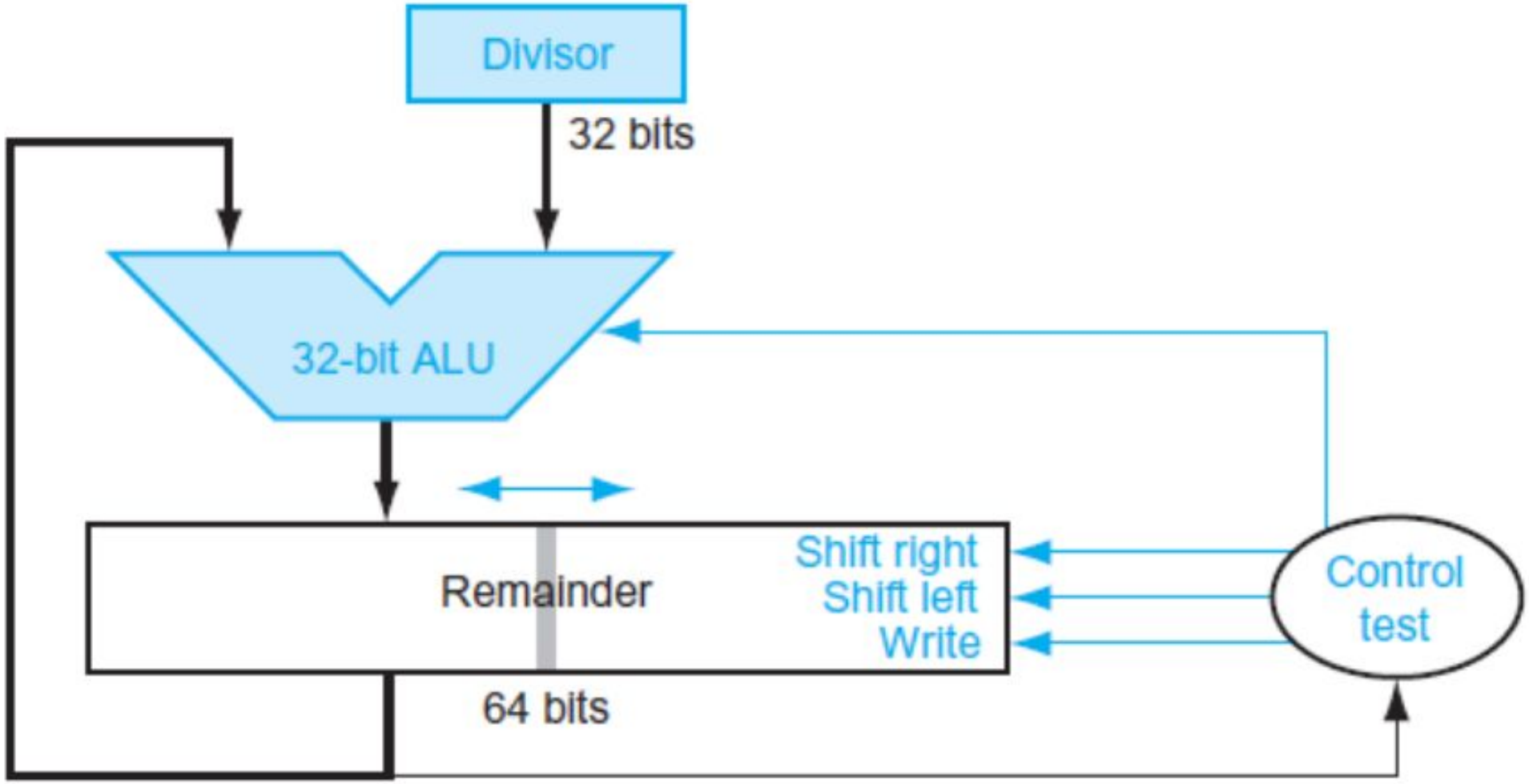


FIGURE **An improved version of the division hardware.** The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in Figure 3.5, the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)

Floating Point Representation

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\ldots_{ten}$ (pi)

$2.71828\ldots_{ten}$ (*e*)

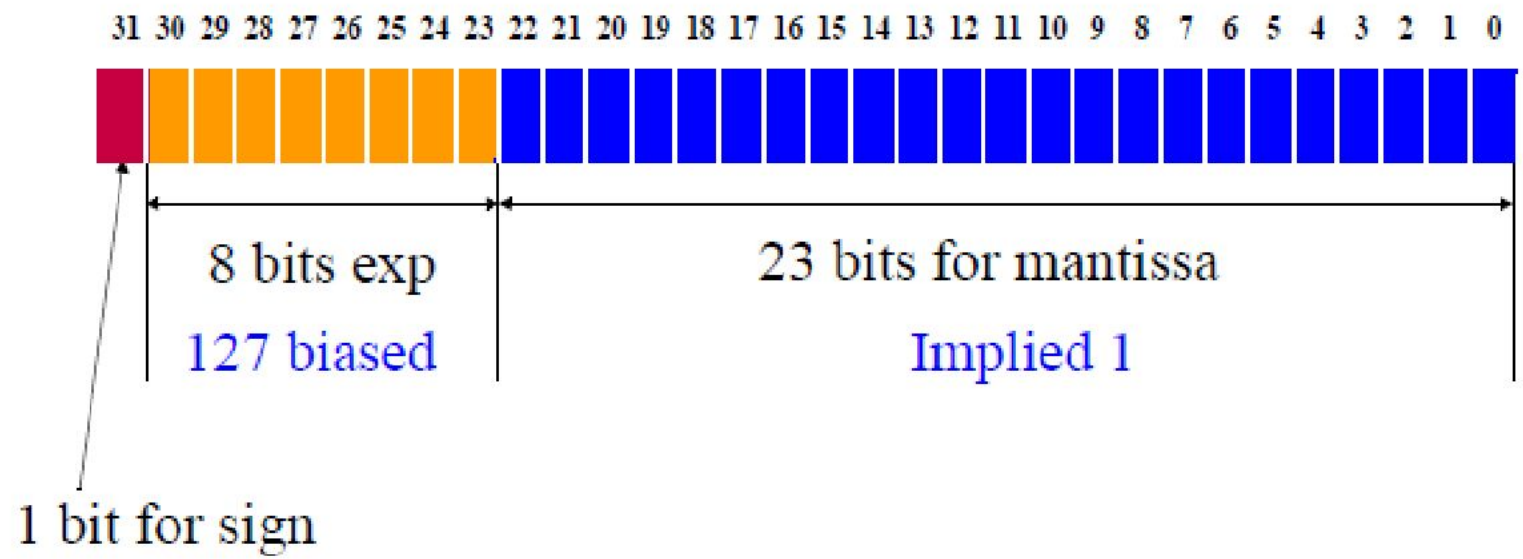
0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)

$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number, which is the usual way to write it. For example, $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not.

IEEE 754 32-bit Single Precision

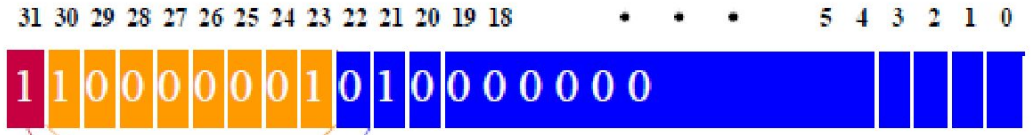
$$(-1)^{S} \times (1+\text{Mantissa}) \times 2^{\text{Exp} - 127}$$



127 Biased Exponent

Original		Biased	
−127 =	1000 0001	−127 =	0000 0000
−126 =	1000 0010	−126 =	0000 0001
....		....	
−1 =	1111 1111	−1 =	0111 1110
+0 =	0000 0000	+0 =	0111 1111
+1 =	0000 0001	+1 =	1000 0000
...		...	
+126 =	0111 1110	+126 =	1111 1110
+127 =	0111 1111	+127 =	1111 1111

Floating Point to Decimal



$(-1)^1 \times (1 + .0100..00) \times 2^{(129 - 127)}$

$= -1 \times (1.025) \times 2^2$

$= -1.25 \times 4$

$= -5.0$

Decimal to Floating Point

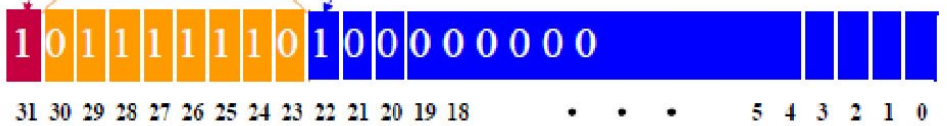
$-0.75 = -3/4 = -3/2^2$;represent in power of two

$= -11_{two}/2^2$;convert to binary

$= -.11_{two}$;shift left to normalize

$= -1.1_{two} \times 2^{-1}$

$= (-1)^1 \times (1 + .1000..00) \times 2^{(126 - 127)}$

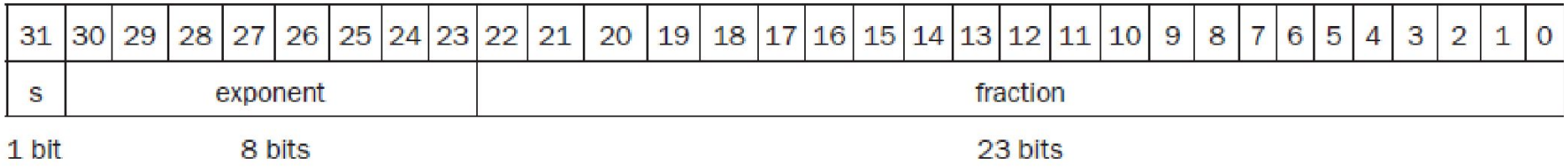


0 00000000 00000000000000000000000 = 0

1 00000000 00000000000000000000000 = -0

0 11111111 00000000000000000000000 = Infinity

1 11111111 00000000000000000000000 = -Infinity



In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

These chosen sizes of exponent and fraction give MIPS computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{ten} \times 10^{-38}$ and numbers almost as large as $2.0_{ten} \times 10^{38}$ can be represented in a computer.

MIPS double precision allows numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the much larger fraction.

Negative exponents pose a challenge to simplified sorting. If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number.

The desirable notation must therefore represent the most negative exponent as $00 \ldots 00_{two}$ and the most positive as $11 \ldots 11_{two}$. This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

overflow (floating-point) A situation in which a positive exponent becomes too large to fit in the exponent field.

underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

double precision A floating-point value represented in two 32-bit words.

single precision A floating-point value represented in a single 32-bit word.

IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and $+1$ is represented by $1 + 127$, or $128_{ten} = 1000\ 0000_{two}$. The exponent bias for double precision is 1023. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

The range of single precision numbers is then from as small as

$$\pm 1.00000000000000000000000_{two} \times 2^{-126}$$

to as large as

$$\pm 1.11111111111111111111111_{two} \times 2^{+127}.$$

An Encoding Example

- Consider the number F = 15335

  $15335_{10} = 11101111100111_2 = 1.1101111100111 \times 2^{13}$

- Mantissa will be stored as: $M = 10111111001110000000000_2$

- Here, EXP = 13, BIAS = 127. ➔ $E = 13 + 127 = 140 = 10001100_2$

0	10001100	10111111001110000000000

466F9C00 in hex

- Consider the number F = -3.75

  $-3.75_{10} = -11.11_2 = -1.111 \times 2^1$

- Mantissa will be stored as: $M = 11100000000000000000000_2$

- Here, EXP = 1, BIAS = 127. ➔ $E = 1 + 127 = 128 = 10000000_2$

1	10000000	11100000000000000000000

40700000 in hex

Floating-Point Representation

EXAMPLE

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

ANSWER

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

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	0	0	0

32 bits

EXAMPLE

Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

ANSWER

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$
\begin{aligned}
(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\
&= -1 \times 1.25 \times 2^2 \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$

Floating Point Addition/Subtraction

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$, where $E1 > E2$ (say).
- Basic steps:
  - Select the number with the smaller exponent (i.e. *E2*) and shift its mantissa right by (*E1-E2*) positions.
  - Set the exponent of the result equal to the larger exponent (i.e. *E1*).
  - Carry out $M1 \pm M2$, and determine the sign of the result.
  - Normalize the resulting value, if necessary.

- Suppose we want to add F1 = 270.75 and F2 = 2.375

  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (2.375)_{10} = (10.011)_2 = 1.0011 \times 2^1$

- Shift the mantissa of F2 right by 8 – 1 = 7 positions, and add:

  1000 0111 0110 0000 0000 0000
  1 0011 0000 0000 0000 0000 000
  1000 1000 1001 0000 0000 0000 0000 000

  Residue

- Result: 1.00010001001×2^8

Subtraction Example

- Suppose we want to subtract F2 = 224 from F1 = 270.75

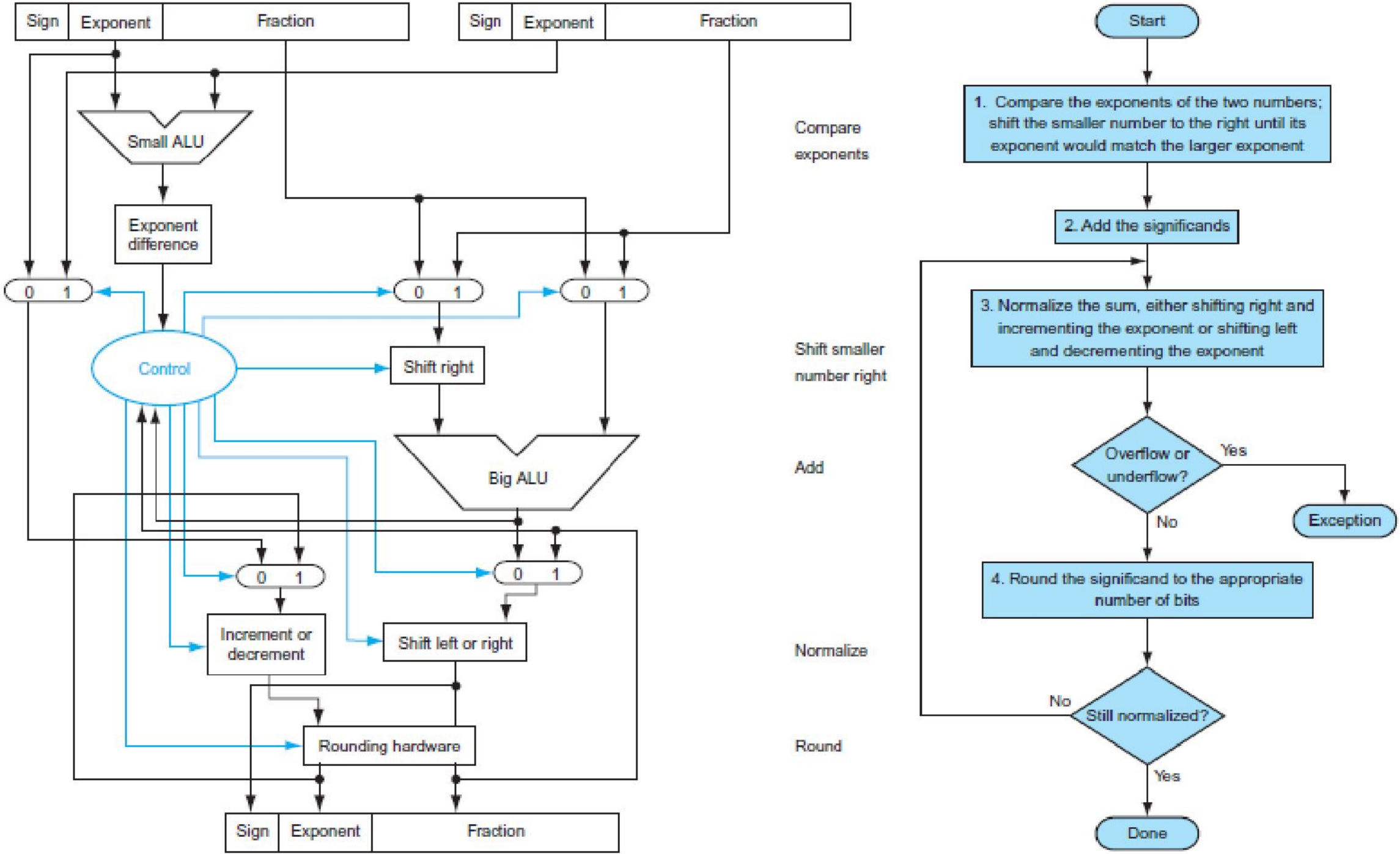
  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (224)_{10} = (11100000)_2 = 1.111 \times 2^7$

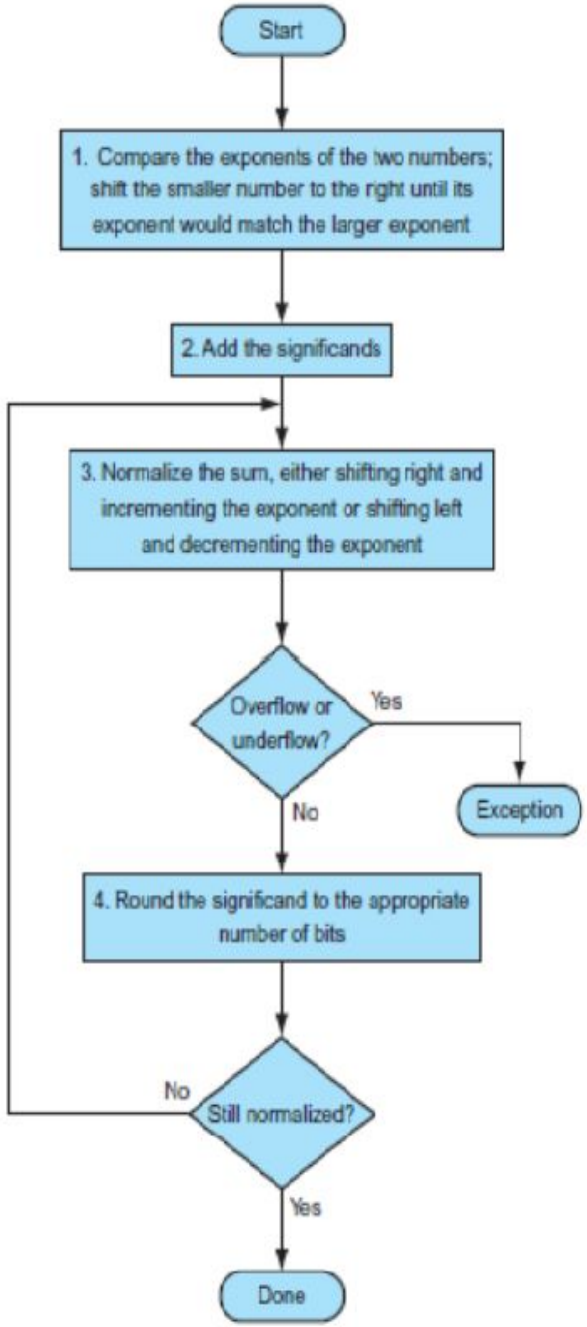
- Shift the mantissa of F2 right by 8 – 7 = 1 position, and subtract:

```
  1000 0111 0110 0000 0000 0000
   111 0000 0000 0000 0000 0000 000
-------------------------------------
  0001 0111 0110 0000 0000 0000 000
```

- For normalization, shift mantissa left 3 positions, and decrement E by 3.
- Result: 1.01110110×2^5


Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Big ALU
0 1
0 1
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round

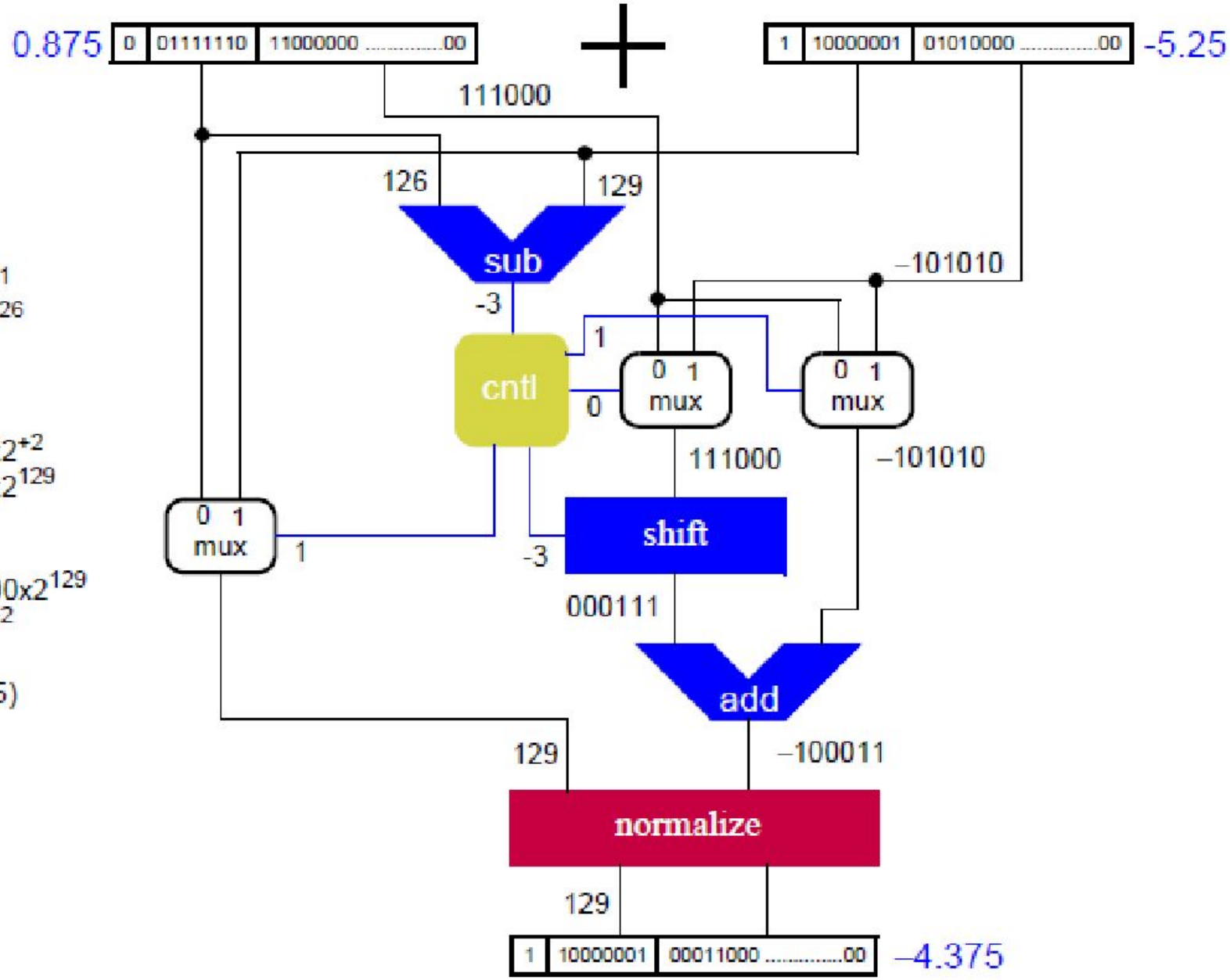
Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done



0.875
$= (-1)^0 \times 0.111$
$= (-1)^0 \times 1.110 \times 2^{-1}$
$= (-1)^0 \times 1.110 \times 2^{126}$

–5.25
$= (-1)^1 \times 101.010$
$= (-1)^1 \times 1.01010 \times 2^{+2}$
$= (-1)^1 \times 1.01010 \times 2^{129}$

0.875 + (–5.25)
$= (-1)^1 \times 1.0001100 \times 2^{129}$
$= -1.0001100 \times 2^{+2}$
$= -100.01100$
$= -(4+0.25+0.125)$
$= -4.375$



Question

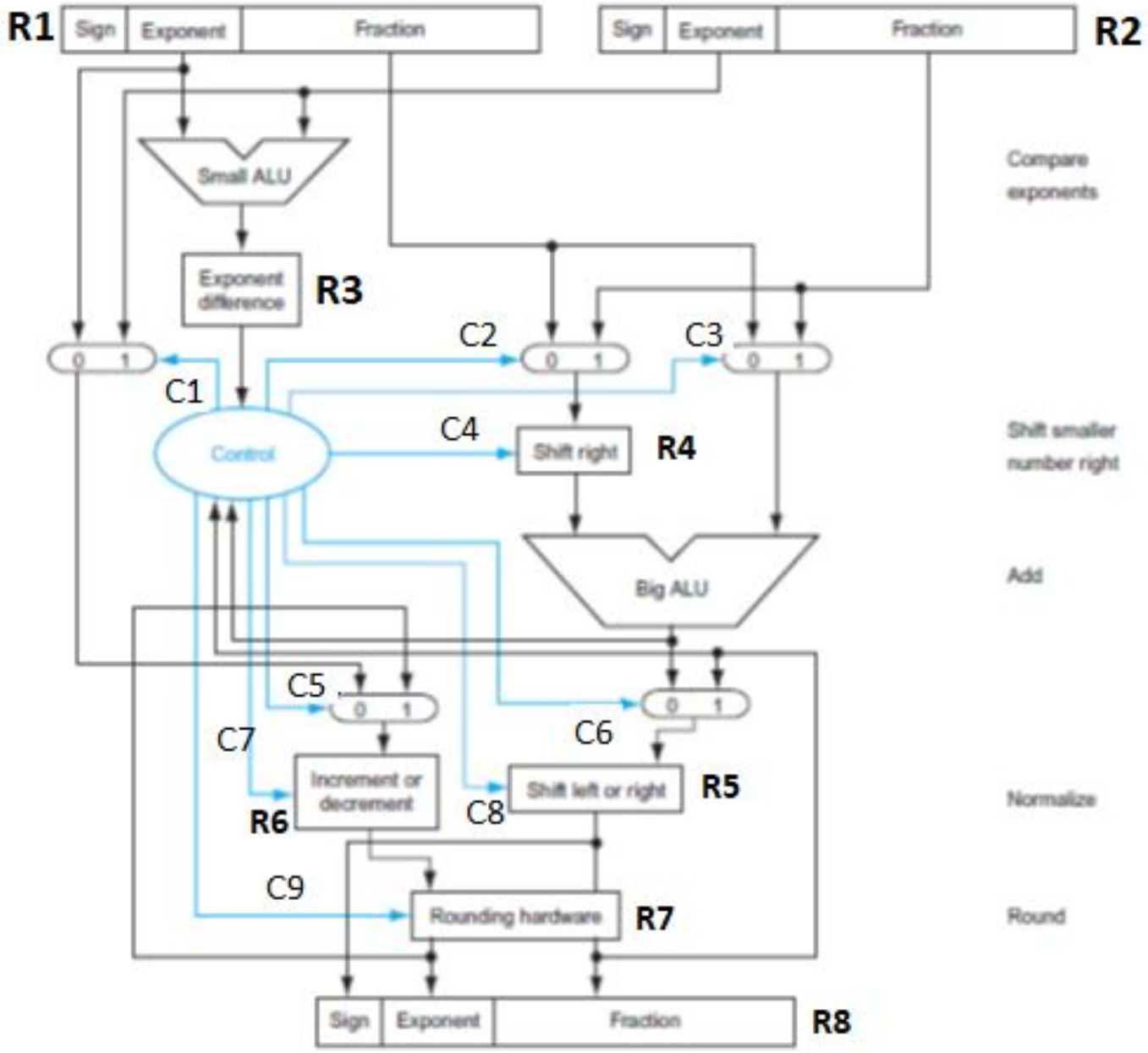
Using the hardware given to the right, show the addition of two numbers in steps as it would take place.

A = 8.625, B = - 7.125

Show the steps in detail. Also show the contents of different registers (R1, .. R8) and control bits (C1, .. C9). It is noted that Control bits

(C1, .. C9) could be either 1 or 0 and you can assign the logic levels for particular operations.

You must present your answer in a Table, given below. (Increase no of columns and rows as required)



Floating-Point Multiplication

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
  - Add the exponents *E1* and *E2* and subtract the *BIAS*.
  - Multiply *M1* and *M2* and determine the sign of the result.
  - Normalize the resulting value, if necessary.

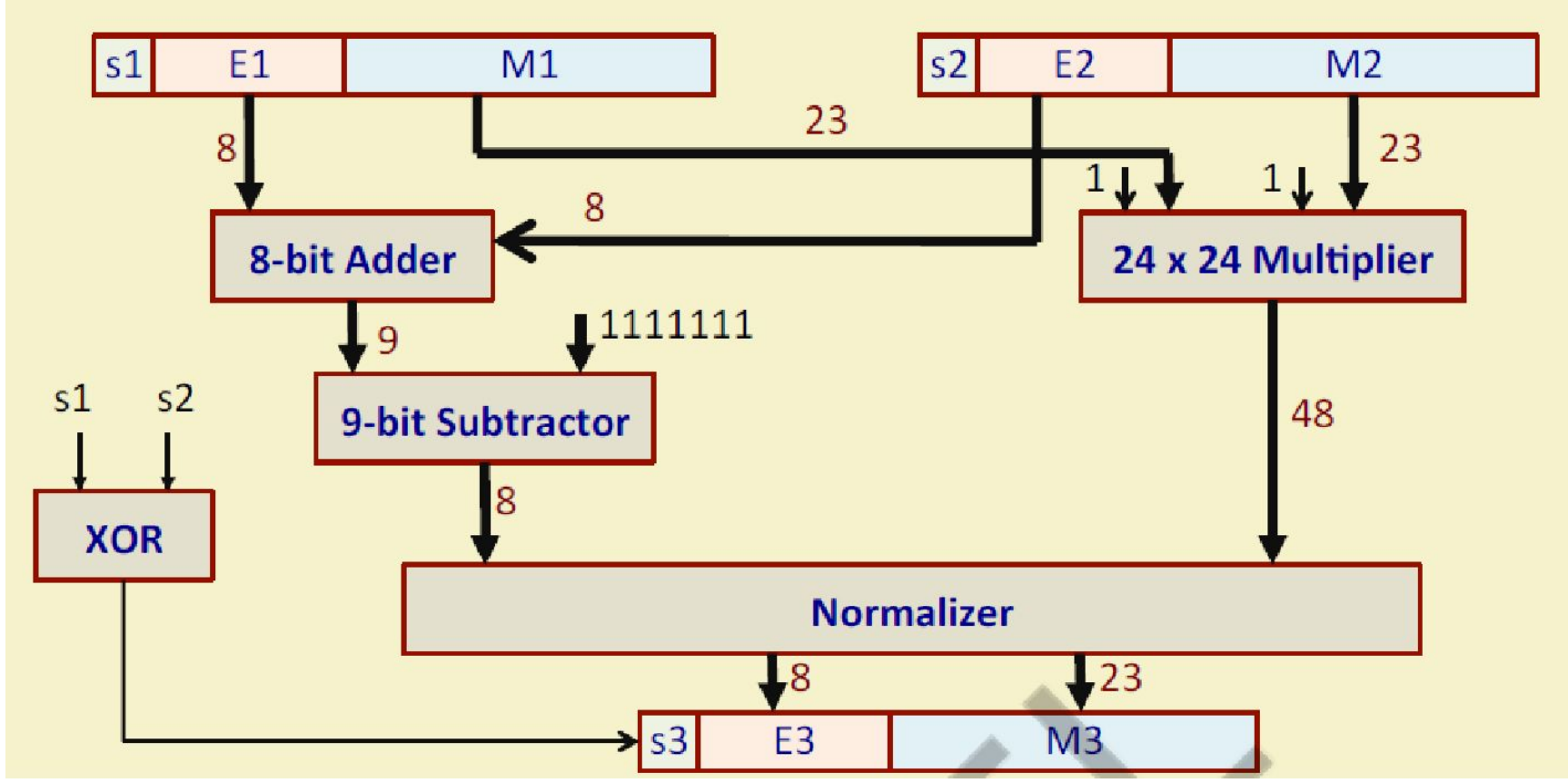
- Suppose we want to multiply F1 = 270.75 and F2 = -2.375

  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (-2.375)_{10} = (-10.011)_2 = -1.0011 \times 2^1$
- Add the exponents: 8 + 1 = 9
- Multiply the mantissas: 1.01000001100001
- Result: $1.01000001100001 \times 2^9$

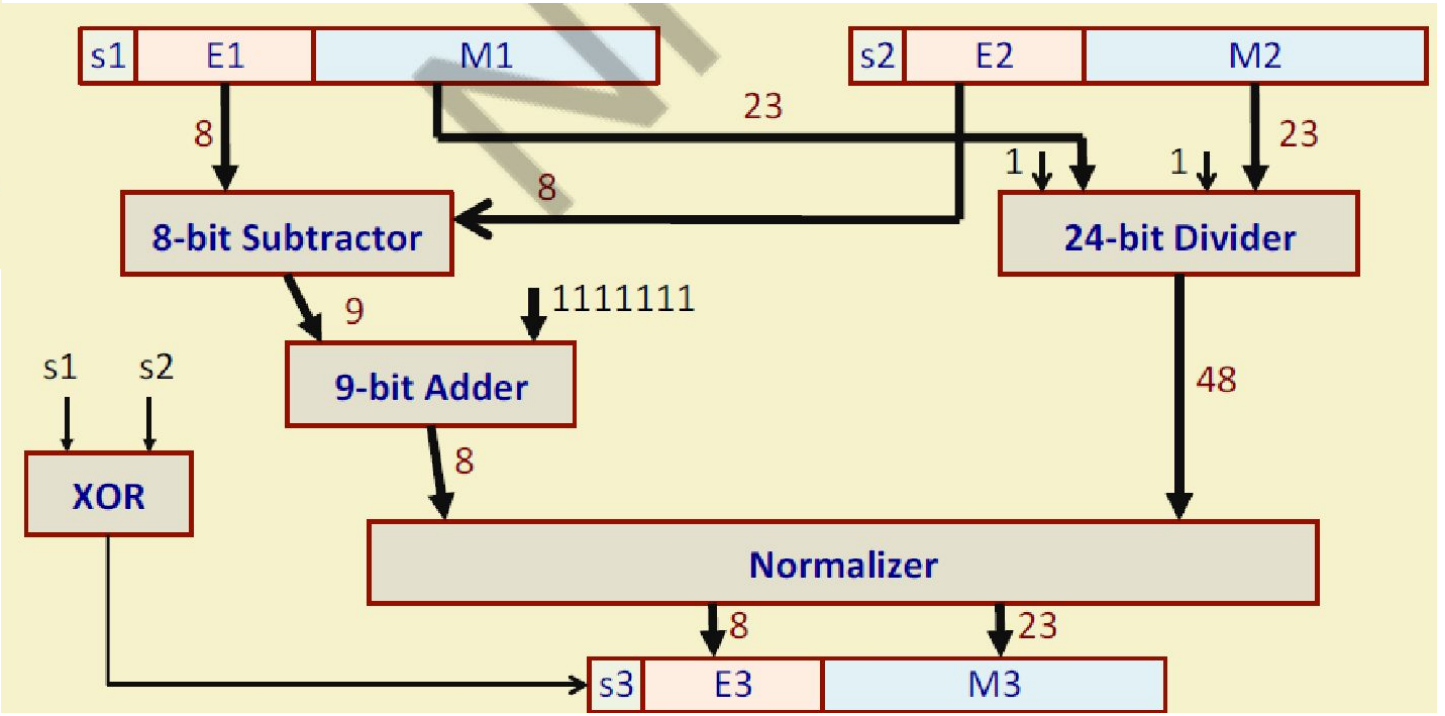
Floating-Point Multiplication

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
  - Add the exponents *E1* and *E2* and subtract the *BIAS*.
  - Multiply *M1* and *M2* and determine the sign of the result.
  - Normalize the resulting value, if necessary.



Floating-Point Division

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
  - Subtract the exponents *E1* and *E2* and add the *BIAS*.
  - Divide *M1* by *M2* and determine the sign of the result.
  - Normalize the resulting value, if necessary.



- Suppose we want to divide F1 = 270.75 by F2 = -2.375

  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

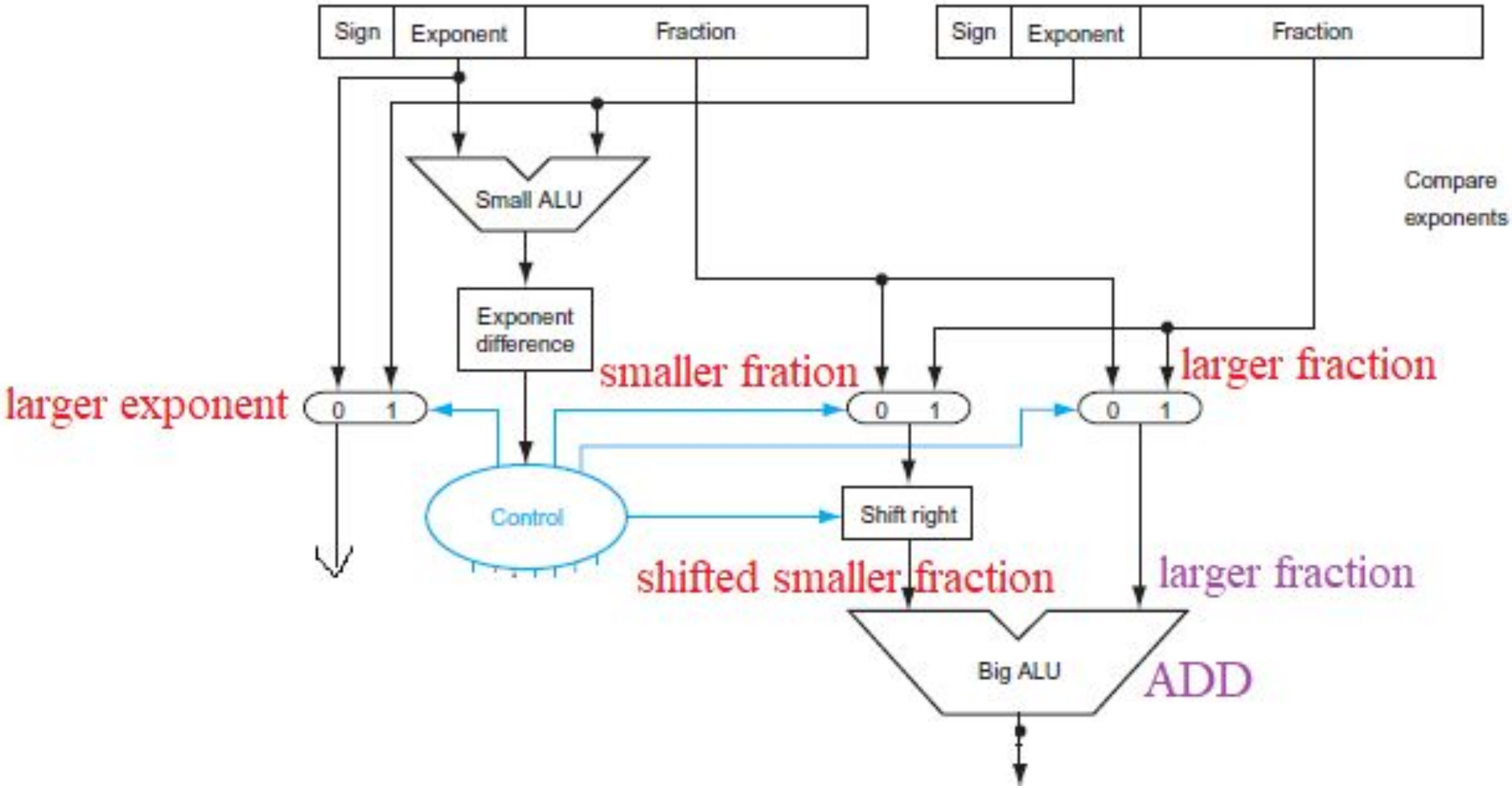
  $F2 = (-2.375)_{10} = (-10.011)_2 = -1.0011 \times 2^1$
- Subtract the exponents: 8 – 1 = 7
- Divide the mantissas: 0.1110010
- Result: 0.1110010×2^7
- After normalization: 1.110010×2^6

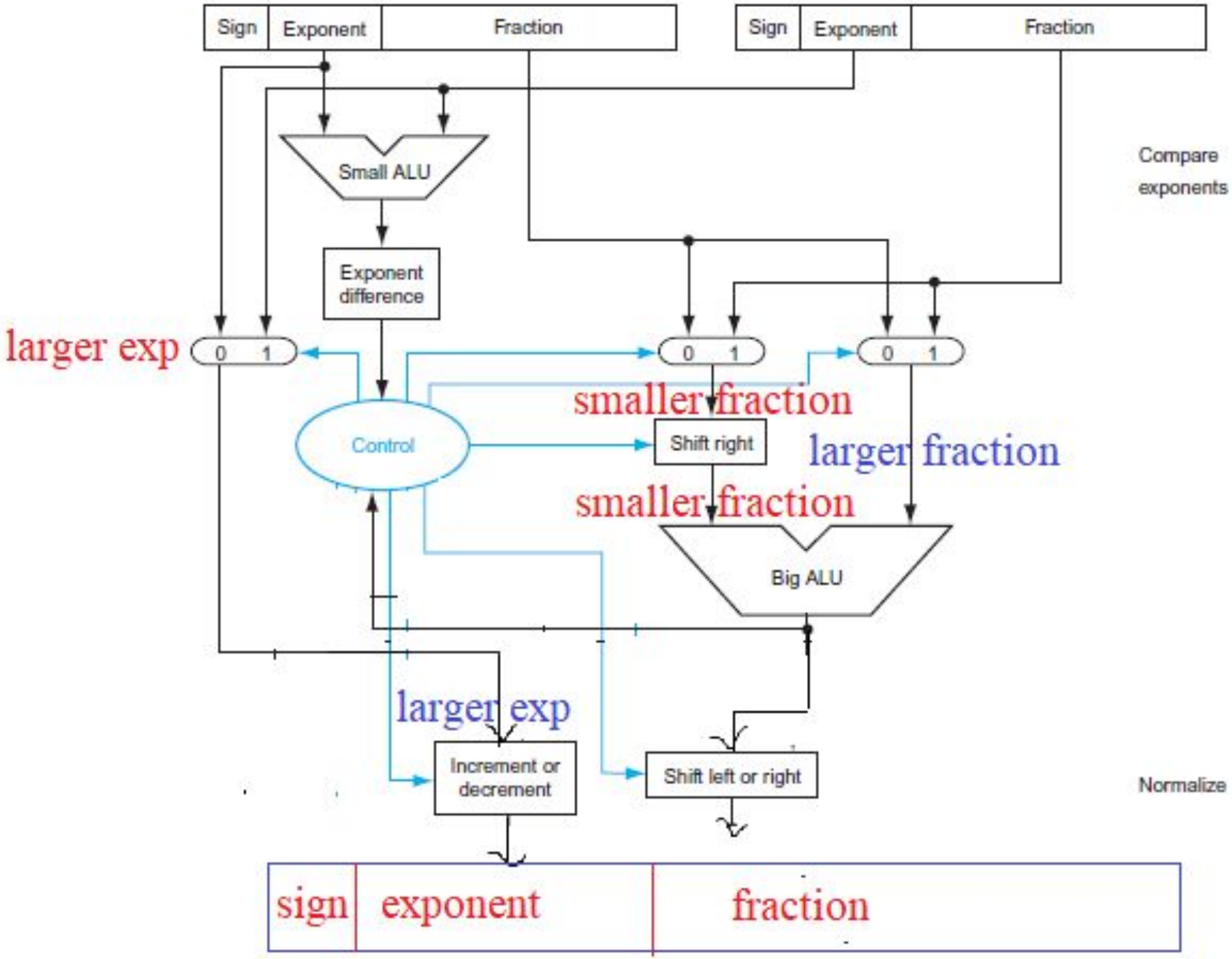
Example: FLOATING POINT ADDITION

$$0.5_{ten} \text{ and } -0.4375_{ten}$$

$$\begin{aligned}
0.5_{ten} &= 1/2_{ten} &&= 1/2^{1}_{ten} \\
&= 0.1_{two} &&= 0.1_{two} \times 2^{0} &&= 1.000_{two} \times 2^{-1} \\
-0.4375_{ten} &= -7/16_{ten} &&= -7/2^{4}_{ten} \\
&= -0.0111_{two} &&= -0.0111_{two} \times 2^{0} &&= -1.110_{two} \times 2^{-2}
\end{aligned}$$

$1.000_{two} \times 2^{-1}$ $-1.110_{two} \times 2^{-2}$




Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Compare exponents
Exponent difference
larger exp
0 1
0 1
0 1
smaller fraction
Control
Shift right
larger fraction
smaller fraction
Big ALU
larger exp
Increment or decrement
Shift left or right
Normalize
sign
exponent
fraction

Binary Floating-Point Addition

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 3.14.

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$\begin{aligned} 0.5_{ten} &= 1/2_{ten} &&= 1/2^{1}{}_{ten} \\ &= 0.1_{two} &&= 0.1_{two} \times 2^{0} &&= 1.000_{two} \times 2^{-1} \\ -0.4375_{ten} &= -7/16_{ten} &&= -7/2^{4}{}_{ten} \\ &= -0.0111_{two} &&= -0.0111_{two} \times 2^{0} = -1.110_{two} \times 2^{-2} \end{aligned}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$0.001_{two} \times 2^{-1} = 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3}$$
$$= 1.000_{two} \times 2^{-4}$$

Since $127 \geq +4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{two} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$1.000_{two} \times 2^{-4} = 0.0001000_{two} = 0.0001_{two}$$
$$= 1/2^{4}_{ten} \qquad = 1/16_{ten} \qquad = 0.0625_{ten}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.